Users Manual

70-Series Microprocessor

National Semiconductor

Publication Number μPG-000001
Order No. μPG-000001
September 1979

70-SERIES MICROPROCESSOR

USER'S MANUAL

© National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051

PREFACE

The 70-Series Users Manual provides a functional description of the architecture and operation of the 70-Series family of single-chip, bus-oriented microprocessors. This family includes the INS8070 and INS8072. The manual also describes the 70-Series assembly language instruction set, the definition and use of macro instructions, and various programming techniques. The material within this publication is up-to-date at the time of publication and is subject to change without notice.

Copies of this and other National Semiconductor publications may be obtained from the National Semiconductor sales office or distributor serving your locality.

Related Publications:

- Series 8000 Microprocessor Family Handbook
- Series/80 BLC 80P Prototyping Package Users Manual
- BLC 80/10 Hardware Reference Manual
- Series/80 Microcomputer System Brochure

National does not assume any responsibility for use of any circuitry described. No circuit patent licenses are implied, and National reserves the right, at any time without notice, to change said circuitry.

CONTENTS

Chapter Page

1 INTRODUCTION TO THE 70-SERIES FAMILY 1-1
1.1 GENERAL DESCRIPTION 1-1
1.2 CONVENTIONS USED IN THIS MANUAL 1-1
1.3 OPERATIONAL FEATURES 1-2
1.4 OVERVIEW 1-2
1.4.1 Timing 1-2
1.4.2 Interrupt/Sense Lines 1-2
1.4.3 Control Flags 1-3
1.4.4 Eight-Bit Data Bus 1-3
1.4.5 Sixteen-Bit Address Bus 1-3
1.4.6 Power Requirements 1-3
1.4.7 Multiprocessing and DMA (Direct Memory Access) 1-3
1.4.8 Control and Data Movement 1-3
1.4.9 Summary of INS807X Registers 1-3
1.5 SOFTWARE FEATURES 1-5
1.6 70-SERIES APPLICATIONS 1-6
1.7 SUPPORT 1-6
1.7.1 MICROBUS 1-6
1.7.2 The MICRO + PROGRAM 1-7
1.7.3 Hardware 1-7
1.7.4 Publications 1-8
1.7.5 Training 1-8
1.7.6 Technical Support Programs 1-9

2 FUNCTIONAL DESCRIPTION 2-1

2.1 INTRODUCTION 2-1
2.1.1 70-Series Features 2-1
2.1.2 Signal Descriptions 2-2
2.2 ARCHITECTURE AND OPERATION 2-5
2.2.1 INS807X Architecture and Internal Registers 2-5
2.2.2 INS807X Operation 2-9
2.2.3 Instruction Set 2-15
2.2.4 Addressing Modes 2-16
2.2.5 Specialized Instructions 2-18
2.2.6 70-Series System Configuration 2-20

3 INSTRUCTION SET 3-1

3.1 ADD EXTENSION TO ACCUMULATOR 3-3
3.2 ADD IMMEDIATE TO ACCUMULATOR 3-3
3.3 ADD IMMEDIATE TO EXTENDED ACCUMULATOR 3-4
3.4 ADD MEMORY TO ACCUMULATOR 3-4
3.5 ADD MEMORY TO EXTENDED ACCUMULATOR 3-5
3.6 ADD MEMORY DIRECT TO ACCUMULATOR 3-6
3.7 ADD MEMORY DIRECT TO EXTENDED ACCUMULATOR 3-6
3.8 AND EXTENSION WITH ACCUMULATOR 3-7
3.9 AND IMMEDIATE WITH ACCUMULATOR 3-7
3.10 AND IMMEDIATE WITH STATUS REGISTER 3-8
3.11 AND MEMORY WITH ACCUMULATOR 3-9
3.12 AND MEMORY DIRECT WITH ACCUMULATOR 3-10
3.13 BRANCH IF ACCUMULATOR NOT ZERO 3-10
3.14 BRANCH IF ACCUMULATOR POSITIVE 3-11
3.15 BRANCH IF ACCUMULATOR ZERO 3-12
3.16 BRANCH IF NOT DIGIT IN ACCUMULATOR 3-12
3.17 BRANCH UNCONDITIONAL 3-13
3.18 CALL 3-14
3.19 DECREMENT AND LOAD ACCUMULATOR 3-16
3.20 DECREMENT AND LOAD ACCUMULATOR FROM MEMORY 3-16
3.21 DIVIDE 3-17

Chapter Page

3.22 EXCLUSIVE OR EXTENSION WITH ACCUMULATOR 3-18
3.23 EXCLUSIVE OR IMMEDIATE WITH ACCUMULATOR 3-18
3.24 EXCLUSIVE OR MEMORY WITH ACCUMULATOR 3-19
3.25 EXCLUSIVE OR MEMORY DIRECT WITH ACCUMULATOR 3-20
3.26 EXCHANGE ACCUMULATOR AND EXTENSION REGISTER 3-21
3.27 EXCHANGE EXTENDED ACCUMULATOR AND POINTER REGISTER 3-21
3.28 INCREMENT AND LOAD 3-22
3.29 INCREMENT AND LOAD DIRECT 3-22
3.30 JUMP TO SUBROUTINE 3-23
3.31 JUMP UNCONDITIONAL 3-24
3.32 LOAD ACCUMULATOR TO EXTENSION REGISTER 3-24
3.33 LOAD ACCUMULATOR TO STATUS REGISTER 3-25
3.34 LOAD EXTENSION TO ACCUMULATOR 3-25
3.35 LOAD EXTENDED ACCUMULATOR TO POINTER REGISTER 3-25
3.36 LOAD EXTENDED ACCUMULATOR TO TEMPORARY REGISTER 3-26
3.37 LOAD IMMEDIATE TO ACCUMULATOR 3-26
3.38 LOAD IMMEDIATE TO EXTENDED ACCUMULATOR 3-27
3.39 LOAD IMMEDIATE TO POINTER REGISTER 3-27
3.40 LOAD IMMEDIATE TO TEMPORARY REGISTER 3-28
3.41 LOAD MEMORY TO ACCUMULATOR 3-28
3.42 LOAD MEMORY TO EXTENDED ACCUMULATOR 3-29
3.43 LOAD MEMORY TO TEMPORARY REGISTER 3-30
3.44 LOAD MEMORY DIRECT TO ACCUMULATOR 3-30
3.45 LOAD MEMORY DIRECT TO EXTENDED ACCUMULATOR 3-31
3.46 LOAD MEMORY DIRECT TO TEMPORARY REGISTER 3-32
3.47 LOAD POINTER REGISTER TO EXTENDED ACCUMULATOR 3-32
3.48 LOAD STATUS REGISTER TO ACCUMULATOR 3-33
3.49 LOAD TEMPORARY REGISTER TO EXTENDED ACCUMULATOR 3-33
3.50 MULTIPLY 3-34
3.51 NO OPERATION 3-34
3.52 OR EXTENSION WITH ACCUMULATOR 3-35
3.53 OR IMMEDIATE WITH ACCUMULATOR 3-35
3.54 OR IMMEDIATE WITH STATUS REGISTER 3-36
3.55 OR MEMORY WITH ACCUMULATOR 3-37
3.56 OR MEMORY DIRECT WITH ACCUMULATOR 3-38
3.57 POP STACK TO ACCUMULATOR 3-38
3.58 POP STACK TO EXTENDED ACCUMULATOR 3-39
3.59 POP STACK TO POINTER REGISTER 3-39
3.60 PUSH ACCUMULATOR TO STACK 3-40
3.61 PUSH EXTENDED ACCUMULATOR TO STACK 3-40
3.62 PUSH POINTER REGISTER TO STACK 3-41
3.63 PUSH AND LOAD POINTER REGISTER IMMEDIATE 3-41
3.64 RETURN 3-42
3.65 ROTATE ACCUMULATOR RIGHT 3-42
3.66 ROTATE ACCUMULATOR RIGHT WITH LINK 3-43
3.67 SEARCH AND SKIP IF CHARACTER MATCH 3-43
3.68 SHIFT ACCUMULATOR LEFT 3-44
3.69 SHIFT ACCUMULATOR RIGHT 3-44
3.70 SHIFT ACCUMULATOR RIGHT WITH LINK 3-45
3.71 SHIFT EXTENDED ACCUMULATOR LEFT 3-45
3.72 SHIFT EXTENDED ACCUMULATOR RIGHT 3-45
3.73 STORE ACCUMULATOR IN MEMORY 3-46
3.74 STORE ACCUMULATOR IN MEMORY DIRECT 3-47
3.75 STORE EXTENDED ACCUMULATOR IN MEMORY 3-47
3.76 STORE EXTENDED ACCUMULATOR IN MEMORY DIRECT 3-48
3.77 SUBTRACT EXTENSION FROM ACCUMULATOR 3-49
3.78 SUBTRACT IMMEDIATE FROM ACCUMULATOR 3-49
3.79 SUBTRACT IMMEDIATE FROM EXTENDED ACCUMULATOR 3-50
3.80 SUBTRACT MEMORY FROM ACCUMULATOR 3-50
3.81 SUBTRACT MEMORY FROM EXTENDED ACCUMULATOR 3-51
3.82 SUBTRACT MEMORY DIRECT FROM ACCUMULATOR 3-52
3.83 SUBTRACT MEMORY DIRECT FROM EXTENDED ACCUMULATOR 3-52

Chapter Page

4 ASSEMBLY LANGUAGE 4-1

4.1 BASIC ELEMENTS 4-1
4.1.1 Character Set 4-1
4.1.2 Data Representation 4-1
4.1.3 Terms 4-1
4.1.4 Expressions 4-4
4.1.5 Reserved Words 4-5
4.2 ASSEMBLER CODING CONVENTIONS 4-6
4.2.1 Label Field 4-6
4.2.2 Operation Field 4-6
4.2.3 Operand Field 4-6
4.2.4 Comment Field 4-6
4.2.5 Identification Sequence Field 4-9
4.2.6 Example of Statement 4-9
4.2.7 Delimiters 4-9

5 ASSEMBLER-DEPENDENT STATEMENTS 5-1

5.1 ASSIGNMENT STATEMENT 5-1
5.2 DIRECTIVE STATEMENTS 5-2
5.2.1 .ASCII Directive 5-3
5.2.2 .ADDR Directive 5-3
5.2.3 .BYTE Directive 5-3
5.2.4 .DBYTE Directive 5-4
5.2.5 Macro-time Looping Directives 5-4
5.2.6 Conditional Assembly Directives 5-4
5.2.7 ERROR Directive 5-6
5.2.8 EXIT Directive 5-6
5.2.9 END Directive 5-6
5.2.10 Conditional Assembly Directive 5-7
5.2.11 INCLD Directive 5-7
5.2.12 LIST Directive 5-8
5.2.13 LOCAL Directive 5-9
5.2.14 Macro Definition Directives 5-10
5.2.15 Delete Macro Directive 5-11
5.2.16 Macro Local Label Directive 5-11
5.2.17 .PAGE Directive 5-11
5.2.18 .SET Directive 5-11
5.2.19 .SPACE Directive 5-12
5.2.20 Symbol Table Value Directive 5-12
5.2.21 TITLE Directive 5-12

6 MACROS 6-1

6.1 INTRODUCTION 6-1
6.2 BASIC MACRO CONCEPTS 6-1
6.3 DEFINING A MACRO 6-2
6.4 CALLING A MACRO 6-3
6.5 USING PARAMETERS 6-4
6.5.1 Macro Definition 6-4
6.5.2 Calling a Macro with Parameters 6-5
6.5.3 Parameters Referenced by Number 6-6
6.5.4 ' ∧ '—Concatenation 6-6
6.6 LOCAL SYMBOLS 6-6
6.7 CONDITIONAL EXPANSION 6-7
6.7.1 .IF, .ELSE, .ENDIF Directives 6-7
6.7.2 .IFC Directive 6-7
6.8 USEFUL DIRECTIVES 6-8
6.8.1 .SET Directive 6-8
6.8.2 .MDEL Directive 6-8
6.8.3 .ERROR Directive 6-8
6.9 MACRO-TIME LOOPING 6-8
6.9.1 .DO and .ENDDO Directives 6-8
6.9.2 .EXIT Directive 6-9
6.9.3 Examples of Macro-Time Loops 6-9
6.10 NESTED MACRO CALLS 6-10
6.11 NESTED MACRO DEFINITIONS 6-11

7 PROGRAMMING TECHNIQUES 7-1

7.1 SCOPE 7-1
7.2 SUBROUTINES 7-1
7.3 TRANSFERING DATA TO SUBROUTINES 7-3
7.4 TEXT PROGRAMMING TECHNIQUES 7-4
7.5 INPUT AND OUTPUT PROGRAMMING TECHNIQUES 7-5
7.5.1 Programmed Input/Output 7-6
7.5.2 Interrupt Input/Output 7-6

8 70-SERIES APPLICATIONS 8-1
8.1 TERMINALS, PERSONAL COMPUTERS, VIEWDATA SETS 8-1
8.2 MULTIPROCESSING SYSTEMS 8-1
8.3 INTELLIGENT PERIPHERAL CONTROLLER 8-2

APPENDICES

Appendix **Page**

A 70-SERIES INSTRUCTION SET A-1
B 70-SERIES FAMILY OP CODE LISTING B-1
C INDEX OF DIRECTIVES C-1
D INS8070-SERIES MICROPROCESSOR FAMILY DATA SHEET D-1

ILLUSTRATIONS

Figure		Page
1-1	70-Series Internal Block Diagram	1-4
2-1	70-Series Pin Configuration	2-4
2-2	INS8072 Configuration, Block Diagram	2-4
2-3	Frequency Control Networks for On-Chip Oscillator	2-7
2-4	INS807X Driver and Receiver Equivalent Circuits	2-8
2-5	Reset Implementation	2-9
2-6	External Memory Read/Write Cycle Timing	2-11
2-7	NHOLD Timing	2-12
2-8	Interrupt Transfer Locations	2-13
2-9	Interrupt Service Timing	2-14
2-10	807X Instruction Format	2-15
2-11	Addressing Mode Specification	2-16
2-12	Memory Locations for the Destination Addresses for Call	2-18
2-13	807X Multiprocessing System	2-21
2-14	807X in a Typical DMA System	2-23
3-1	Memory Locations of One-Byte Call Starting Addresses	3-15
4-1	Relationship of Terms	4-2
4-2	Sample Coding Form	4-7
4-3	Sample Program Listing	4-8
6-1	Statement Insertion	6-1
7-1	Programmed Input/Output	7-6
7-2	Interrupt Input/Output Initiation	7-7
8-1	General Configuration for Terminals, Personal Computers and Viewdata	8-1
8-2	Multiprocessing Systems	8-1
8-3	Typical 807X to System Connection	8-2
8-4	Interrupt and Acknowledge	8-3

TABLES

Table		Page
3-1	70-Series I/O Signal Description	3-2
3-2	Symbols and Notation	3-1
4-1	Arithmetic and Logical Operators	4-5
5-1	Summary of Assembler Directives	5-2
5-2	List of Options	5-8
A-1	70-Series Instruction Set	A-1
B-1	Alphabetic Index of Directives	B-1

Chapter 1

INTRODUCTION TO THE 70-SERIES FAMILY

1.1 GENERAL DESCRIPTION

The 70-Series family of single-chip, bus-oriented microprocessors is designed for compact, cost-reduced control, data acquisition, and processing applications. Up to 2.5K bytes of ROM and 64 bytes of RAM are available on-chip.

The flexibility of the bus architecture and special peripherals permit the development of highly integrated systems, ranging from the two-chip minimum system with six channels of D/A conversion plus 23 bits of discrete I/O, to sophisticated multiprocessing systems in which each 70-Series element may become a node of an array.

The 70-Series family consists of the following components:

- The INS8070, minimum system containing 64 bytes of RAM and no on-chip ROM
- The INS8072, containing 64 bytes of RAM and 2.5K bytes (20K bits) of ROM

Fabricated using National's standard N-channel silicon gate, depletion load MOS technology, the 70-Series family of devices are each housed in a standard 40-pin dual-in-line package, and require only a single 5-volt power supply.

1.2 CONVENTIONS USED IN THIS MANUAL

The following conventions are used in this manual:

- For brevity, the designator 807X is used to refer to both members of the 70-series family, i.e., the INS8070 and INS8072.
- Positive logic convention is used throughout. A logic '1' or high signal corresponds to a more-positive voltage level. A logic '0' or low signal corresponds to a more-negative voltage level. All signal names beginning with 'N' denote complemented signals that are asserted or activated by a logic '0'. Otherwise, signals are asserted by a logic '1'.
- The "X'" or "0" preceding a value denotes the hexadecimal numbering system.

1.3 OPERATIONAL FEATURES

The 70-Series CPU is a programmable 8-bit parallel processor. A 16-bit address bus and 8-bit bidirectional data bus are provided and both are TRI-STATE®. Full multiprocessing and DMA (Direct Memory Access) control protocols are implemented on-board. Major features of the 70-Series family are summarized below.

- Simplified programming, with multiple addressing modes, including program counter-relative, immediate data, indexed, auto-indexed, implied and single-byte subroutine calls
- Stack operation, with a comprehensive set of 16-bit instructions, including load, store, add, subtract, multiply, divide, exchange, shift, and stack push/pop
- Unique features including an ASCII-to-BCD conversion in one instruction and the character search instruction, both of which are especially useful for high-level language programming
- Large system capability, with address capability up to 65K bytes of memory
- Handshake bus-access control on-chip, for multiprocessing and DMA operations
- System flexibility, with the ability to interface with memory and standard INS8080 peripherals
- On-chip clock generation, two interrupt/sense inputs, and three user-accessible control-flag outputs
- Low system cost, using standard memories and standard peripheral components
- High reliability, incorporating the inherent reliability of LSI devices, low system component count and low power consumption
- Full software development support via National's STARPLEX Microprocessor Development System
- Comprehensive user support, supplied by field and factory applications engineers, technical training courses, and National's Microprocessor Users Group

1.4 OVERVIEW

The INS807X is a self-contained general-purpose microprocessor designed for ease of implementing stand-alone, DMA (Direct Memory Access), and multiprocessor applications. Communications between the INS807X and external memory/peripheral devices are affected via a 16-bit address bus, an 8-bit bidirectional data bus. (See figure 1-1), and appropriate control and flag signals.

1.4.1 Timing

For applications where timing is not particularly critical, only a resistor-capacitor network is required for the clock—all other timing circuits are an integral part of the chip. If the application requires precision clock control, the RC network can be replaced with a crystal. No external timing circuits (clock drivers, phase splitters, buffers, and so fourth) are required in either case.

1.4.2 Interrupt/Sense Lines

The 70-Series CPU has two sense lines: Sense A and Sense B. Under program control, either or both sense lines can be examined and a decision can be made based on the logic state of one or both inputs. The sense lines serve a dual function in that they may also be used as interrupt inputs: when Sense A and B are used as interrupts, the INS807X can monitor asynchronous events while running other programs.

1.4.3 Control Flags

Discrete control of events that occur external to the microprocessor are provided by three control flags (F1, F2 and F3). These TTL-compatible outputs are built into the 807X chip. Under software control, each flag output can produce pulses or fixed logic levels. Thus, lamps, motors, relays, and other similar devices can be serviced as required by the application. Since any combination of the flags can be set with a simple series of instructions, there is a significant saving in external decoding logic.

1.4.4 Eight-Bit Data Bus

The eight-bit data bus provides a means of bidirectional data transfer between the internal registers of the CPU, and standard memory components or peripheral devices external to the CPU. Valid input data are expected to be on the bus at the end of the active low read-strobe (NRDS) time, whereas output data are valid on the bus at the active low write-strobe (NWDS) time.

NOTE: In applications that use Direct Memory Access, data transfers between external memory and a peripheral can be made without entering the data port of the INS807X; that is, the INS807X does not actively participate in the actual DMA transfer of data.

1.4.5 Sixteen-Bit Address Bus

The 70-Series CPU provides 16 dedicated address lines (AD 00 through AD 15) that are internally latched—an applications feature that can save much external logic.

1.4.6 Power Requirements

A 5-Volt ±5% power supply is all that is required.

1.4.7 Multiprocessing and DMA (Direct Memory Accessing)

One of the most notable features of the 70-Series microprocessor family is the built-in bus allocation logic for both multiprocessor and DMA (Direct Memory Access) applications. All the necessary control logic and timing signal generation are provided on-chip. Three signals are provided to implement these protocols:

- NBREQ—Bus Request
- NENIN—Enable In
- NENOUT—Enable Out

1.4.8 Control and Data Movement

The INS807X is initialized by pulling the reset (NRST) input low. The reset input must be allowed to return high for the device to start operation.

NOTE: The NRST (reset) input must remain high during INS807X operation.

All data enter and exit the INS807X via the 8-bit bidirectional data bus. When the processor is started, the program counter (PC) is incremented and the first instruction byte is fetched from the address specified by the contents of the PC - memory location 0001_{16}. The instruction is decoded and executed under control of the programmable logic array (PLA).

1.4.9 Summary of INS807X Registers

The INS807X contains six 16-bit registers and six 8-bit registers. The programmer has access to eight of these registers.

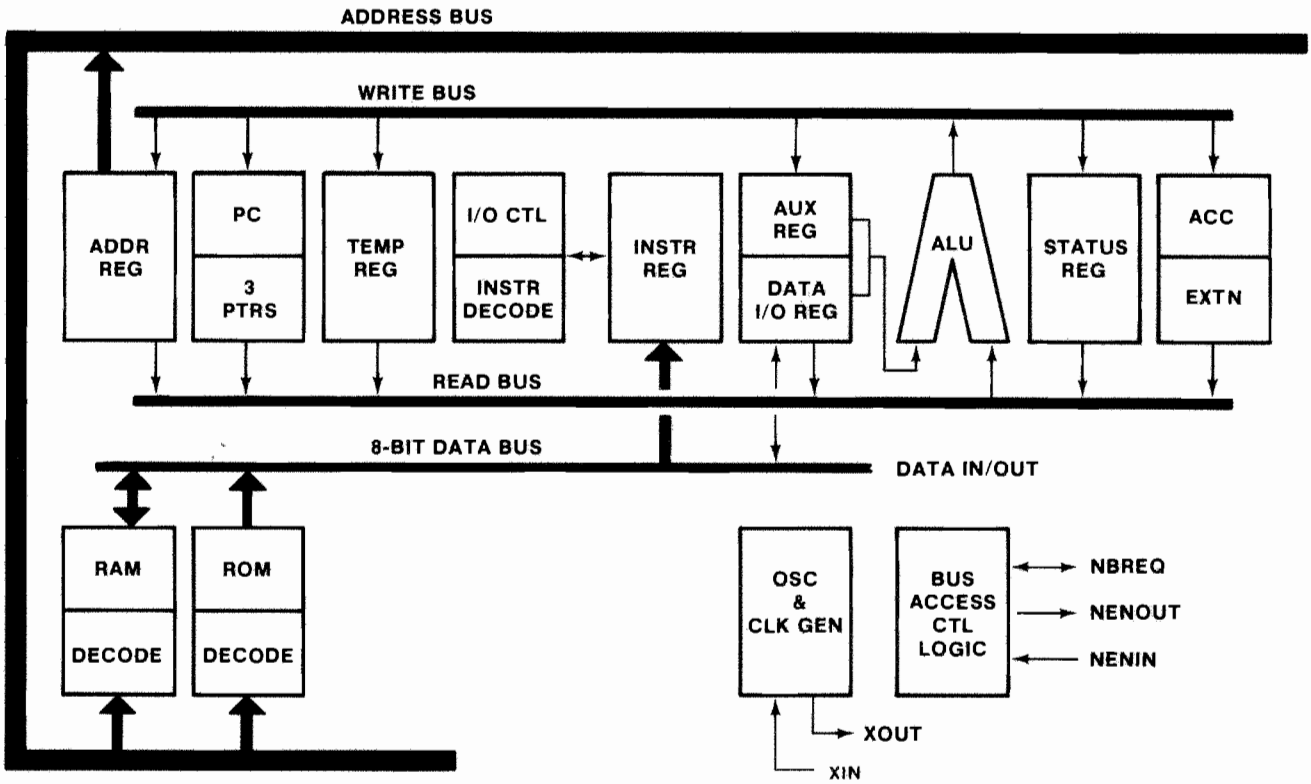


FIGURE 1-1. 70-Series Family Simplified Block Diagram

18-2

1.4.9.1 Programmer Accessible Registers

The program controllable registers are a combination of 8-bit and 16-bit registers as follows:

a. Program Counter (PC) - 16 bits wide

b. Temporary Register (T) - 16 bits wide

c. Pointer Registers (SP, P2, P3) - 16 bits wide

d. Accumulator (A) - 8 bits wide

e. Extension Register (E) - 8 bits wide

f. Status Register (S) - 8 bits wide

NOTE: The accumulator may also be combined with the extension register to form the 16-bit extended accumulator (EA).

1.4.9.2 Nonaccessible Registers

The following registers are not directly accessible to the programmer

a. Address Register - 16 bits wide

b. Instruction Register - 8 bits wide

c. Auxiliary Register - 8 bits wide

d. Data Input/Output Register - 8 bits wide

1.5 SOFTWARE FEATURES

The instruction set of the 807X provides arithmetic, logical, branch, shift, and other operations between the accumulator, memory, and the registers. The basic instructions combined with the wide range of addressing modes provide a total of 192 different instructions. The instruction set is described in chapter 3; the basic instructions are listed below:

LOAD
STORE
EXCHANGE REGISTERS
ADD
SUBTRACT
MULTIPLY
DIVIDE
AND
OR
EXCLUSIVE-OR
SHIFT RIGHT
SHIFT RIGHT WITH LINK
ROTATE RIGHT
ROTATE RIGHT WITH LINK
SHIFT LEFT
SEARCH AND SKIP IF CHARACTER MATCHED
BRANCH IF NOT DIGIT
PUSH
PUSH AND LOAD IMMEDIATE
POP
BRANCH UNCONDITIONAL
BRANCH IF POSITIVE
BRANCH IF ZERO
BRANCH IF NOT ZERO
JUMP UNCONDITIONAL
JUMP TO SUBROUTINE
CALL
RETURN
LOAD PC
EXCHANGE PC
INCREMENT AND LOAD
DECREMENT AND LOAD
NO OPERATION

1.6 70-SERIES APPLICATIONS

The 807X can be used in many diverse applications. The chip may be used in a minimum configuration that might include a few switches for control, a read-only memory for implementing instructions, and a few indicators for monitoring purposes. On the other hand, a maximum system might include multiple processors, several input-output peripherals, and external read/write as well as read-only memory, and a full-featured control panel. Whatever the application, the 807X and its supporting products include everything needed to develop, debug, and implement a system.

A partial listing of 70-Series applications is shown below. By no means are the uses, features, and benefits all-inclusive: they are indicative of how applications-oriented the INS807X is and how cost-effective it can be in solfing control problems.

- Test systems and instrument control
- Machine tool control
- Personal computers
- Terminals
- Intelligent ROMs
- Small business machines
- Word processing systems
- Educational systems
- Multiprocessor systems
- Process Control
- Traffic control
- Laboratory instrumentation
- Sophisticated games
- Automotive controller
- Appliances

1.7 SUPPORT

National supports its microprocessor-based systems with a variety of software, hardware development systems, applications engineering services, and training services.

To ensure full support capability, all of the ancillary devices required are supplied by National and have been thoroughly tested in systems produced by National. The 70-Series family is supported by an easy-to-use development system, called STARPLEX™. STARPLEX is a completely integrated hardware and software development system for use by the 70-Series family of microprocessors. Various components and software are also available to support the 70-Series family from experimentation to final production, including an in-system emulator (ISE).

1.7.1 MICROBUS™

The MICROBUS defines a universal system bus for communication among National's microprocessors, memories, and peripherals. The MICROBUS concept simplifies the design engineer's job by giving common definitions to all data, address, control, and timing signals commonly used in interfacing microprocessors and peripheral components. This releases the design engineer from the drudgery of translating signal names or worrying about signal compatibility, and gives him more time to focus on the most important aspects of his design. the MICROBUS concept is described in detail in the Series 8000 Microprocessor Family Handbook.

1.7.2 The MICRO+ Program

MICRO+ is a comprehensive quality assurance program for users who cannot perform incoming inspection of microprocessors or those who do not wish to do so but still need higher reliability levels. Users who specify MICRO+ processed parts will find that the program offers the following benefits:

- Reduces the cost of reworking assembled boards
- Reduces field failures
- Simplifies system checkout
- Reduces system downtime
- Eliminates the need for independent test laboratories
- Eliminates incoming electrical inspection

1.7.3 Hardware

The 70-Series processors are supported by a variety of hardware/software products available from National Semiconductor's extensive Series/80 product line. This section presents a brief description of the devices available.

1.7.3.1 Memory Boards

Expansion MOS RAM boards of 16K, 32K, 48K, and 64K bytes and expansion ROM, PROM, EPROM cards in 16K or 32K capacities are available.

1.7.3.2 Input/Output and Memory Expansion Boards

Input and output port capacity can be expanded with boards that provide different configurations of I/O ports and expansion memory.

1.7.3.3 BLC 604 System Chassis and BLC 614 Expansion Board Cage

The BLC 604 System Chassis and BLC 614 Expansion Board Cage provide immediate and economical solutions for housing, interconnection, and expansion of Series/80 format series. Stand-alone BLC 604 or 614 Board Cage holds up to four Series/80 boards in a molded chassis with interconnection backplane, bus signal terminating networks, and connectors.

1.7.3.4 RMC 660 System Chassis

The RMC 660 System Chassis supports the Board Level Computer and full complement of Series/80 expansion boards. The chassis includes board cage for eight boards, seven-inch standard RETMA chassis, and BLC 665 heavy duty power supply. It also includes power-fail detect for orderly power-down sequence, and has current-limited outputs with over-voltage protection. Power on-off and reset switches on the front panel are connected to the system bus for external system control.

1.7.3.5 BLC 635 Power Supply

Ready-made, low-cost power is provided by the BLC 635 power supply. It supports most configurations of BLC memory, and I/O expansion boards. DC power is provided on mating cables for direct connection to the BLC 604 System Card Cage. The BLC 635 senses AC power failure and generates a TTL signal for orderly system power shutdown. It has current-limit outputs with over-voltage protection, and 100/115/200/230 VAC input voltage; 50/60 Hz input frequency.

1.7.3.6 BLC 665 Heavy Duty Power Supply

The BLC 665 supplies approximately twice the rated current of the BLC 635.

For additional information, refer to the following National Semiconductor publications:

- Series/80 BLC 80P Prototyping Package Users Manual
- BLC 80/10 Hardware Reference Manual
- Series/80 Microcomputer System Brochure

1.7.4 Publications

Publications are available covering the various devices manufactured by National Semiconductor. The available literature is grouped in the following categories:

- Literature Index
- Handbooks
- Manuals
- Linear Applications Handbook
- Databooks
- Guides
- Product Selection Guides
- Briefs
- Individual Application Notes
- Individual Data Sheets

1.7.5 Training

National Semiconductor operates two microprocessor training centers: in Santa Clara, Calif., and in Lexington, Mass. Each training center is fully equipped and professionally staffed to provide students with a good mix of hardware/software theory and hands-on laboratory experience.

The following courses are offered:

- The Microcourse — designed for the technician, engineer, or manager who has little or no experience with programmable microprocessor devices; covers examples of architecture for a microprocessor, parallel and serial I/O, interval timer, and interrupt controller.
- STARPLEX™ User Course — designed for the individual who has recently obtained, or is considering the purchase of a STARPLEX™ Development System.
- Complex Peripherals — covers the capabilities, limitations, and interfacing of microprocessor programmable peripheral devices, and demonstrates the programming of these devices.
- COPS — designed for the individual who has purchased or is evaluating the Computer Oriented Processor and development system for use in control applications.
- 70-Series — discusses hardware and software design techniques for the 807X family of microprocessors, including multiprocessing and use of specialized instructions.

1.7.6 Technical Support Programs

National Semiconductor has the strongest on-the-scene technical support team — in the U.S. and abroad — of any semiconductor manufacturer. Our large network of independent sales representatives and franchised distributors is backed by our Field Application Engineers (FAE's) and microprocessor applications engineers. The FAE's are available domestically and internationally to offer on-site technical assistance, and are equipped technically to help analyze your application, translate your needs into a viable hardware/software configuration, and then follow it through to system delivery. The microprocessor applications engineers are National's home-base technical-support specialists who support the FAE's in the field, and who help you use your microprocessor most effectively; they are always available to answer specific technical questions regarding the use of National's microprocessor and peripheral components.

Chapter 2

FUNCTIONAL DESCRIPTION

2.1 INTRODUCTION

This chapter provides detailed information on the 70-series family of processors. The information within this chapter is intended to assist the designer by furnishing descriptions of the operating characteristics of the devices and typical interconnection data.

2.1.1 70-Series Features

The 70-Series family of processors is intended for use in systems requiring the economy of a single chip, the flexibility of multiprocessing bus architecture and the power of 16-bit arithmetic operations.

The first members of the 70-Series family include:

INS8070 - 64 Bytes of RAM, no ROM
INS8072 - 64 Bytes of RAM, 2.5K ROM

The 70-Series family comprises bus-oriented microprocesors with on-board ROM and RAM and built-in multiprocessing logic. Low-cost systems containing multiple processors may be constructed, as the CPU's may occupy the same bus and still run at full speed since their external bus requirements are confined to the occasional access of data. Such systems include terminals, intelligent peripherals and multiprocessing systems (e.g., the solution of polynominals in real-time).

The 70-Series family is fabricated using National's standard N-channel, silicon gate depletion load, MOS technology. All the devices are housed in standard 40-pin, dual-in-line packages. Some of the features inherent in the 70-Series processors are as follows:

- On-Board ROM
- On-Board RAM (Executable)
- 8-Bit Data Bus
- 16-Bit Address Bus (64K Addressing Capability)
- Comprehensive Set of 8- and 16-Bit Arithmetic, Logic and Stack Manipulation Instructions
- Hardware 16 x 16-Bit Multiply (37μs) and Divide (42μs)
- Built-in Multiprocessing and DMA Logic
- Interfaces with Standard INS8080 Peripherals
- On-Chip Clock Generation
- Single Instruction Character Search and Single Instruction ASCII-to-Decimal Conversion
- Full Hardware and Software Development Systems
- Single 5-Volt Supply
- MICROBUS™ Compatible

2.1.2 Signal Descriptions

Communication between the 70-Series family and external memory and peripheral devices is via a 16-bit address bus, an 8-bit bidirectional data bus and control lines. INS807X Read and Write strobe outputs (NRDS and NWDS) indicate when valid input/output data are present on the 8-bit data bus. The remaining input/output signals for the INS807X are dedicated to initialization, clock generation, bus management, interrupt request, input/output cycle extension, and user-definable interface functions. Table 2-1 provides the signal descriptions for the INS807X. The table is divided into input signals, output signals, and bidirectional signals, and furnishes the signal mnemonic and functional name. Figure 2-1 illustrates the 70-Series pin configuration.

Table 2–1. 70-Series I/O Signal Description

INPUT SIGNALS	
Reset (NRST)	Active low reset input. This pin initializes the INS807X by resetting the program counter (PC) and the stack pointer (SP) to zero. All bits in the status register (except SA and SB) are also reset to zero. This input is buffered by T²L compatible Schmitt trigger, allowing slow rise and fall times.
Hold (NHOLD)	Active low external memory cycle extend input for slow memories and slow peripherals. Also used for single memory cycle execution. NHOLD affects READ/WRITE cycles to external memory only.
Sense A/Interrupt A (SA/INTA)	Provides a T²L compatible Schmitt trigger buffered sense input which sets/resets the Sense A bit (SA) in the status register. When interrupts are enabled (IE = 1 in the STATUS register) this pin also acts as a trailing edge-triggered interrupt input.
Sense B/Interrupt B (SB/INTB)	Similar operation as SA/INTA. Since SA/INTA has priority over SB/INTB, if both arrive simultaneously, Interrupt A will be serviced first.
Enable Input (NENIN)	Active low bus enable input to the on-chip bus allocation logic. For DMA and multiprocessing applications this logic allows bus access to be granted or denied to the 807X. Devices which can share the bus include other INS807X processors, INS8060 processors, DP8350 CRT Controller and DMA logic. NENIN affects 807X operation as follows: 1) If NENIN is high, 807X sets NENOUT high and is denied access to the bus. 2) If NENIN is low and INS807X is holding NBREQ low, INS807X sets NENOUT high and is granted access to the bus. 3) If NENIN and NBREQ are both low and 807X is not holding NBREQ low, 807X sets NENOUT low, and is denied access to the bus.
POWER	VCC = +5V VSS = GND

Table 2-1. 70-Series I/O Signal Description (Continued)

OUTPUT SIGNALS	
Address Bus (A_{15}-A_0)	TRI-STATE address outputs for all external memory READ and WRITE operations. These outputs are normally in the high-impedance state except during external memory READ and WRITE operations.
Write Data Strobe (NWDS)	TRI-STATE active low WRITE strobe output. NWDS goes low during an external memory WRITE operation. NWDS is in the high-impedance state when an external memory WRITE operation is not in progress.
Read Data Strobe (NRDS)	TRI-STATE active low READ strobe output. NRDS goes low during an external memory READ operation. NRDS is in the high-impedance state when an external memory READ operation is not in progress.
Enable Output (NENOUT)	Active low Bus Enable output from the on-chip bus allocation logic. 807X controls NENOUT as follows: 1) If NBREQ is low but INS807X is not holding it low, NENOUT = NENIN. 2) If NBREQ is low and 807X is holding it low, NENOUT = high. 3) If NBREQ = high, NENOUT = high.
Flags 1, 2, 3 (F_1-F_3)	Flag outputs which can be set high/low by writing into the corresponding flag bits of the status register.
BIDIRECTIONAL SIGNALS	
Bus Request (NBREQ)	NBREQ is a bidirectional line that serves as a bus status/request for the INS807X and any other devices sharing the data bus. The NBREQ pin must be externally connected to V_{CC} through a pullup resistor. When NBREQ acts as an input to the INS807X, a logic high indicates that the data bus (D0-D7) is available; a logic low signifies the data bus is unavailable. When NBREQ acts as an output from the INS807X, a low indicates that the INS807X is requesting access to the data bus; a high indicates that the data bus is available to other devices.
Data Bus (D_0-D_7)	Bidirectional TRI-STATE data bus (input/output). These lines are normally in the high-impedance state except during external memory READ and WRITE operations. They provide input data during external memory READ; they provide output data during external memory WRITE.
Clock Inputs (X_{IN}, X_{OUT})	On-chip clock generator input/output. The on-chip clock generator will operate with external crystal or RC circuit connected between X_{IN} and X_{OUT} The on-chip clock generator may be disabled by supplying a T^2L level clock input on X_{IN}. In all modes a buffered T^2L level clock output is always available at X_{OUT}.

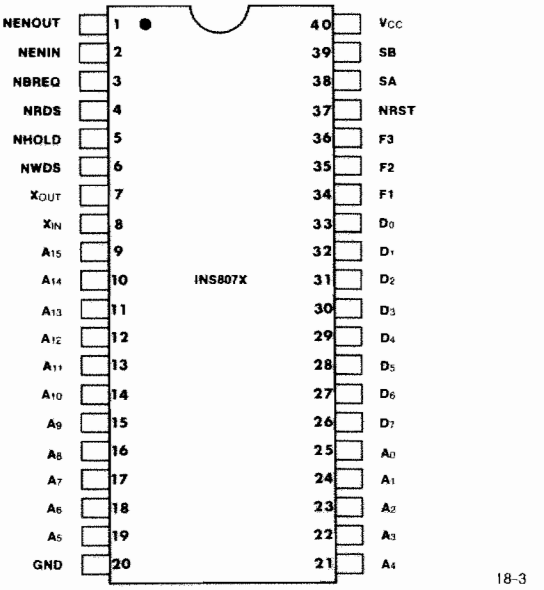
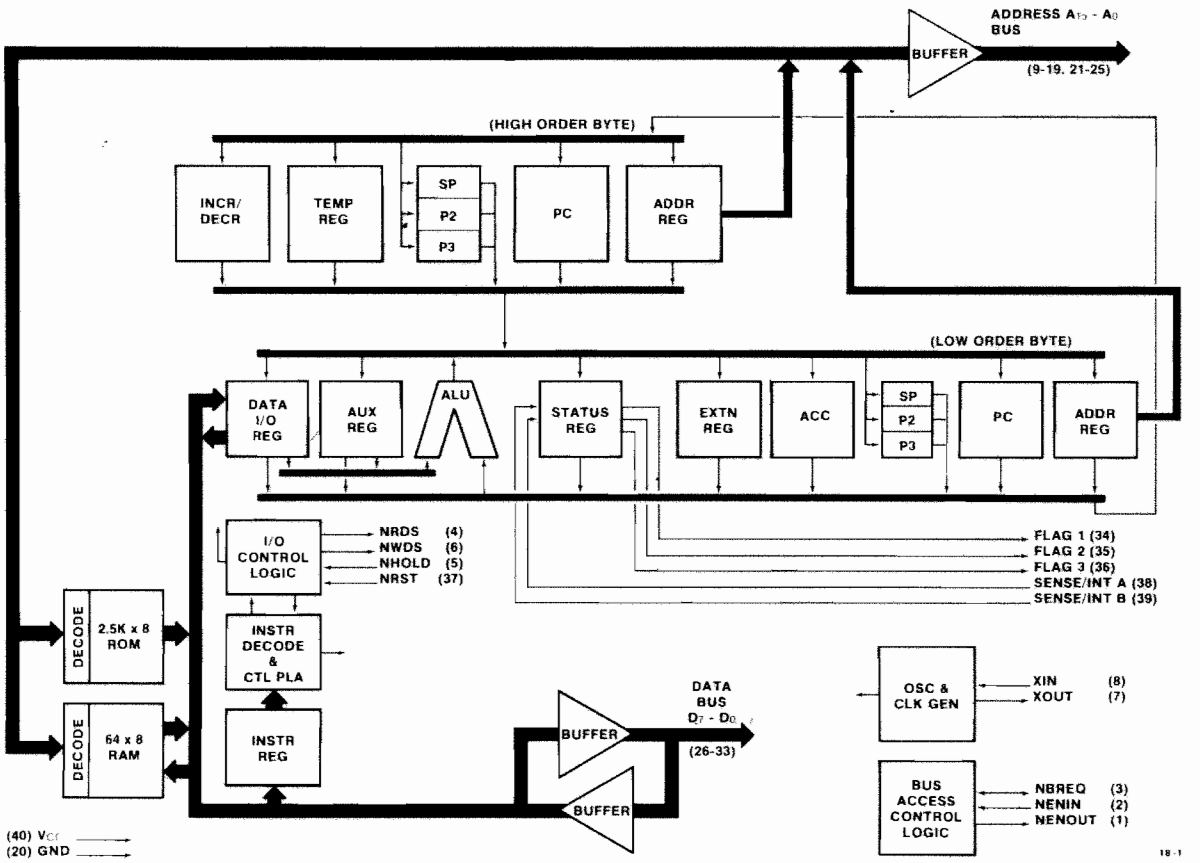


Figure 2 – 1. 70-Series Pin Configuration



NOTE: Applicable pinout numbers are included within parentheses.

Figure 2 – 2. INS8072 Configuration, Block Diagram

2.2 ARCHITECTURE AND OPERATION

The INS8070 and INS8072 are members of a single-chip, general purpose microprocessor family containing zero and 2.5K bytes, respectively, of on-chip ROM. All 70-Series devices contain 64 bytes of on-board RAM. The 70-Series devices may be used in stand-alone, direct memory access and multiprocessor applications. Figure 2-2 is a block diagram of the 70-Series family architecture and device pinouts. For convenience, an INS8072 is depicted to show the on-chip ROM and associated decode circuit.

2.2.1 INS807X Architecture and Internal Registers

As shown in figure 2-2 the INS807X contains six, 16-bit registers and six, 8-bit registers; their functions are described in the following paragraphs.

2.2.1.1 Program Counter: (PC)

The program counter is a 16-bit register that contains the address of the instruction to be executed. The program counter is incremented before each instruction fetch. The program counter also may be replaced by the contents of a specified memory location if the last instruction was a branch, jump, or jump to subroutine instruction. This causes execution to continue from the program specified location rather than from the next sequential location.

If the instruction being executed is a subroutine call, or if an interrupt occurred and interrupts are enabled, the processor counter contents are pushed onto the stack and are subsequently retrieved (popped) from the stack when control is returned to the main program.

The contents of the PC may also be exchanged with the contents of the extended accumulator (EA).

2.2.1.2 Pointer Registers: (SP, P2, P3)

The three pointers, SP, P2, and P3, are 16-bit registers that may be modified under program control. A pointer that has been loaded with a given address can reference up to 256 memory or I/O bytes without being modified. This is accomplished using pointer relative addressing.

The stack pointer, SP, may be used as a general-purpose pointer and/or to point to the top of a RAM-resident stack which is implicitly referenced by the PUSH, POP and PLI instructions. Under program control, this stack may be located anywhere in memory and may be of arbitrary length.

In addition to the stack referenced by the stack pointer (SP), pointers P2 and P3 may also be used to reference other stacks. This is accomplished using auto-indexed addressing. The ability to define and manipulate multiple stacks under program control can improve program efficiency to a considerable degree.

2.2.1.3 Accumulator: AC

The 8-bit accumulator register (AC) is the primary working register of the INS807X. The AC contains the result of all single-byte (8-bit) arithmetic and logic operations. Its contents may also be shifted, rotated, exchanged with the extension register (E), and moved on and off the stack using the PUSH and POP instructions.

2.2.1.4 Extension Register: E

The 8-bit extension register (E) is used for temporary storage, to perform arithmetic and logic operations in conjunction with the accumulator (AC) and combined with the accumulator to form the extended accumulator (EA) for 16-bit operations.

2.2.1.5 Extended (Double Precision) Accumulator Register: EA

The 16-bit extended accumulator register (EA) is the primary double-byte working register of the INS807X. It is formed by concatenation of the 8-bit accumulator register (AC) and the 8-bit extension register (E). All double-byte words are transferred between (EA) and memory in two, single-byte transfers. The (EA) register contains the results of all double-byte arithmetic operations, (add, subtract, multiply, divide). Its contents may also be shifted, pushed (popped) on (off) the stack, stored (loaded) to (from) the pointers and PC and exchanged with the contents of the pointers and PC. When combined with the accumulator to form the EA, the extension register contains the high-order bits.

2.2.1.6 Temporary Register: T

The 16-bit temporary register (T) is used for temporary storage of double-byte words. It is also used to hold the 16-bit multiplier or the 16-bit divisor in multiplication/division. The T register may be loaded directly from memory and its contents may also be transferred to/from the (EA) register.

2.2.1.7 Status Register: S

The 8-bit status register (S) is used to store the following status bits: carry/link (CY/L), 2's complement overflow (OV), sense B (SB), sense A (SA), flags (F1, F2, F3) and the interrupt enable bit (IE).

Status register bits 1, 2, 3 are buffered and connected to the flag output pins F1, F2 and F3. These pins are particularly useful for direct I/O control and for execution of serial I/O operations in conjunction with the SA and SB sense inputs.

(CY/L) goes high to indicate a carry output following execution of an addition/subtraction instruction.

(OV) goes high to indicate 2's complement overflow following execution of an addition/subtraction instruction.

SA and SB are Schmitt trigger buffered inputs which may be tested under software control. They form bits 4 and 5 of the status register and are read-only bits. All other status register bits may be both written to or read from. In addition, all bits may be set and reset, except SA, SB, using the "AND S" and the "OR S" instructions, respectively.

The Interrupt Enable bit (IE), in the Status Register may be set/reset under program control by the OR S and AND S instructions. When set high, IE enables interrupts A and B. When IE is low, both interrupts are disabled. Following a hardware reset via the NRST pin, IE is automatically set low, disabling both interrupts.

The bit functions of the status register are shown below.

7	6	5	4	3	2	1	0	Bit positions
CY /L	OV	SB	SA	F3	F2	F1	IE	flags

NOTE: All status register bits except for SA and SB are latched. When set to a specified state under program control, they will remain in that state until again modified under program control.

2.2.1.8 Instruction Register

This instruction register is not accessible to the programmer. During the fetch cycle of each instruction, this register is loaded with the 8-bit instruction op code retrieved from memory.

2.2.1.9 Auxiliary Register and Data I/O Register

These 8-bit registers are not accessible to the programmer. They are used for temporary storage of input/output bytes received or transmitted over the 8-bit bidirectional data bus.

2.2.1.10 Address Register

The 16-bit address register is not accessible to the programmer. It is used for temporary storage of the 16-bit address transmitted during a memory READ/WRITE cycle.

2.2.1.11 Arithmetic and Logic Unit

The arithmetic and logic unit (ALU) provides the data-manipulation capability that is an essential feature of any microprocessor. The operations provided by the ALU include AND, OR, XOR, increment, decrement, add, subtract, multiply, divide, shift and rotate.

2.2.1.12 Timing Control

All timing signals required for proper operation of the 70-Series processors are provided by an on-chip clock generator.

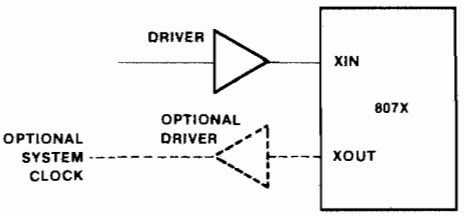
Two control pins, XIN and XOUT, permit the frequency of the on-chip oscillator to be controlled by any of the following methods:

1. Leave the XOUT pin unterminated and drive the XIN pin with an externally generated TTL clock that conforms to the parameters shown in figure 2-3A, the frequency of the oscillator will be equal to the frequency of the external clock input.

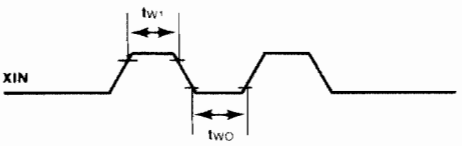
2. Connect a resistor-capacitor feedback network between the XIN and XOUT pins and GND as shown in figure 2-3B.

3. Connect a crystal oscillator with a low-pass filter network between the XIN and XOUT pins and GND as shown in figure 2-3C for a frequency above 1 megahertz or figure 2-3D for a frequency 1 megahertz or below. With this method, the frequency of the on-chip oscillator is equal to the resonant frequency of the crystal. The low-pass filter prevents unwanted harmonic oscillations.

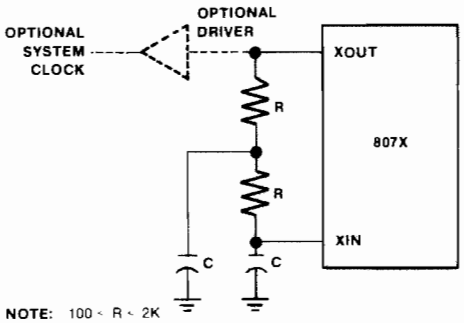
A. External Clock Input



EXTERNAL CLOCK PARAMETERS



B. Register Capacitor Feedback Network



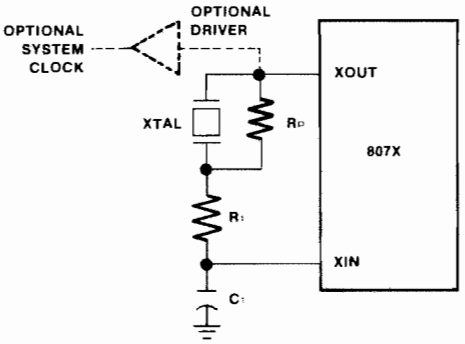
NOTE: 100 < R < 2K

Approximate Oscillator Frequency vs RC Time Constant

$$t = \frac{1}{f} = 0.725\ RC + 353$$

where: R is in KΩ
C is in pf
t is in ns

C. Crystal with Low Pass Filter (Above 1 MHz)

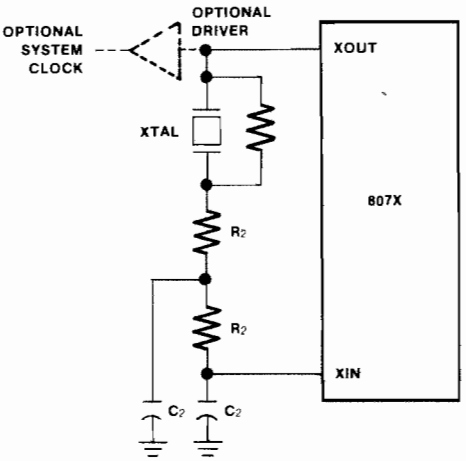


Typical values for Crystal with Low Pass Filter Network

Crystal	Rp	C1	R1
2 MHz	100kΩ	56pF	1kΩ
3.58 MHz	100kΩ	27pF	1kΩ
4 MHz	100kΩ	27pF	1kΩ

XTAL is parallel resonant with maximum series resonance equal to 1kΩ

D. Crystal with Low Pass Filter (1 MHz or Below)



NOTE: The values of R & C are determined by the operating frequency and crystal type

Figure 2-3. Frequency Control Networks for On-Chip Oscillator

18-4

In addition to illustrating appropriate frequency-control networks for the on-chip oscillator, figures 2-3A through 2-3D also show how an optional driver may be used to derive a system clock from the oscillator signal present at the XOUT pin.

The time interval of a microcycle, T_C, is four times the period of the oscillator. Thus:

$$T_C = 4 \frac{1}{f_{osc}} = 4 \frac{1}{f_{res}} = 4 \frac{1}{f_{XIN}}$$

where:

T_C = period of on-chip or external oscillator
f_{osc} = frequency of on-chip oscillator
f_{res} = resonant frequency of crystal connected between XIN and XOUT pins.

2.2.1.13 Drivers and Receivers

Equivalent circuits for INS807X drivers and receivers are shown in figure 2-4. All inputs have static charge protectin circuits consisting of an RC filter and voltage clamp. These devices should be handled with care, as the protection circuits can be destroyed by excessive static charge.

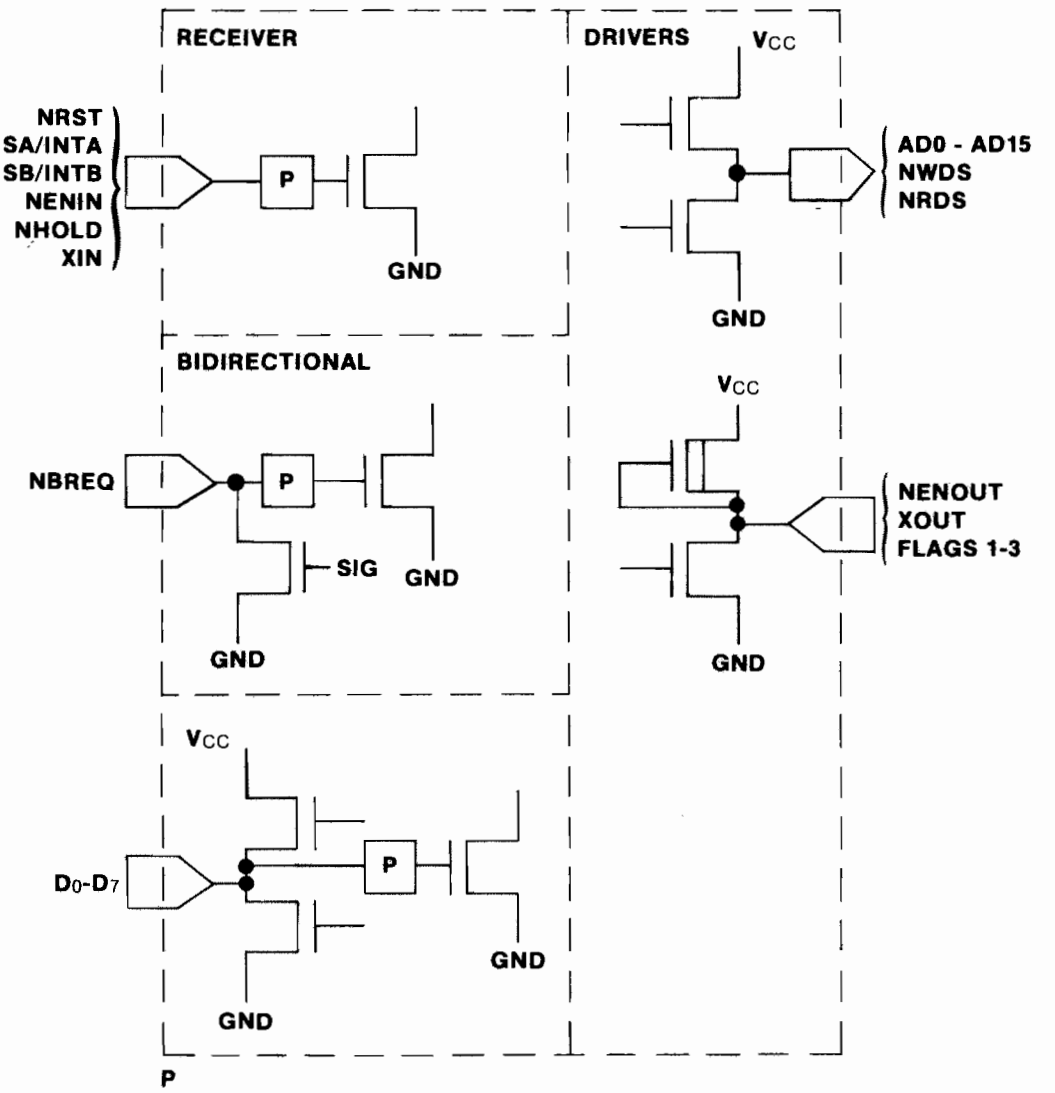


Figure 2 – 4. INS807X Driver and Receiver Equivalent Circuits 18-5

2.2.1.14 Initialization

The recommended power-up initialization sequence for the 807X is as follows:

1. Apply power (GND and V_{CC}) and set NRST low. When NRST goes low the following occurs:

   a) Any in-process operations are automatically aborted.
   b) The READ/WRITE (NRDS/NWDS) strobes, address bus and data bus are set to the TRI-STATE™ (**high Z**) condition.
   c) The Program Counter (PC), Stack Pointer (SP), and Status Register (S) are automatically cleared.

2. Ample time should be allowed for the oscillator and internal clocks to stabilize. The minimum time that NRST must remain low is 8 T_C, where T_C is the period of one microcycle. (NOTE: For a crystal-controlled oscillator, 250 ms are typically required for crystal warmup and stabilization.)

3. To commence program execution, NRST must be set high. The first instruction will be fetched from location 0001_{16}. This will occur within 13 T_C after NRST has gone high.

Normal program execution will continue as long as NRST remains high. In order to allow this signal to have slow rise/fall times without any adverse affects, NRST is buffered by a T^2L compatible Schmitt Trigger. This allows the inexpensive RC-power on reset circuit of figure 2-5 to be used to initialize the 807X.

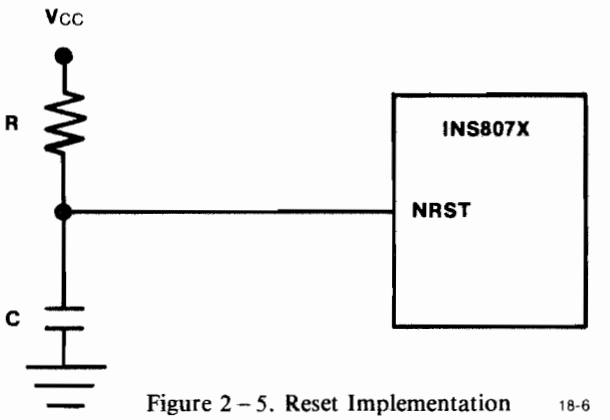


Figure 2 – 5. Reset Implementation 18-6

2.2.2 INS807X Operation

The operation of the INS807X processor consists of repeatedly accessing or fetching instructions from the program stored in internal or external memory and executing the operations specified by the instructions. These two steps are performed under the control of an internal microprogram. Microprogram storage is provided by an on-chip programmable logic array (PLA).

2.2.2.1 Parallel Data Transfers

Parallel data transfers occur during each instruction fetch and during the ensuing read/write cycle associated with execution of each instruction. The 807X accesses its internal memory without exercising the external bus. All read/write operations external to the INS807X are conducted via the data and address buses. This implies that peripheral devices can be memory mapped, that is, treated as memory locations, thus simplifying programming.

This unified bus structure differs from other single-chip microprocessors that have one instruction type (input/output class) for communication with peripheral devices and another instruction type (memory reference class) for communication with memories. The advantage of the approach taken by the 807X is that a wider variety of instructions (the entire memory-reference class) is available for communications with peripherals. Thus, the LD and ST (Load and Store) instructions can be used for basic transfers, the ILD and DLD (increment/decrement and load) instructions can be used for indexing peripheral registers, and the remaining memory reference instructions can be used, as required, for one-step retrieval and processing of peripheral input data.

2.2.2.2 Data Transfer and Control

Data transfers to and from the 807X need not be synchronized, or slaved, to a particular timing sequence; devices with widely different data rates and all using common system buses can be serviced by the processor. A data transfer between the 807X and memory peripherals (with access times of 650 nanoseconds, or less, in systems that use a 4MHz crystal) is a simple and straight-forward process. The input/output sequence proceeds as follows:

- Bus request from the 807X
- Request granted by internal or external logic control
- Address valid
- Data valid and either input or output, as appropriate

Valid data must appear on the data bus before the read strobe makes a low-to-high transition. When using a fast memory, valid data are always present before the read strobe goes inactive; thus, there is no need to extend the read-strobe interval.

2.2.2.3 Input/Output Cycle

Once INS807X has control of the system buses, the actual input/output cycle begins. The functions of data reading, and data writing are implemented, respectively, by the read strobe (NRDS), and the write strobe (NWDS). Note that the NBREQ signal is reset high at the end of the input/output cycle to indicate that the system buses are now freed for use by the highest-priority requesting device.

When INS807X has control of the bus it generates the NRDS or NWDS strobe, respectively, to initiate a data-input (read) or data-output (write) operation. For a read operation, input data are strobed into INS807X from the 8-bit bus at the trailing edge of the NRDS strobe. For a write operation, INS807X places valid output data on the 8-bit bus at the leading edge of the NWDS strobe. After resetting the NRDS or NWDS strobe to complete the data transfer, INS807X then resets the NBREQ signal to indicate that the system buses are free for use by another controller.

2.2.2.4 READ/WRITE Cycle Timing

The 70-Series processor can execute READ and WRITE operations involving internal memory or external memory. All internal memory READ/WRITE operations are executed in one microcycle. All external memory READ/WRITE operations are executed in two microcycles.

Figure 2-6 shows the timing of Read and Write cycles involving external memory.

2.2.2.5 External Bus Access Control

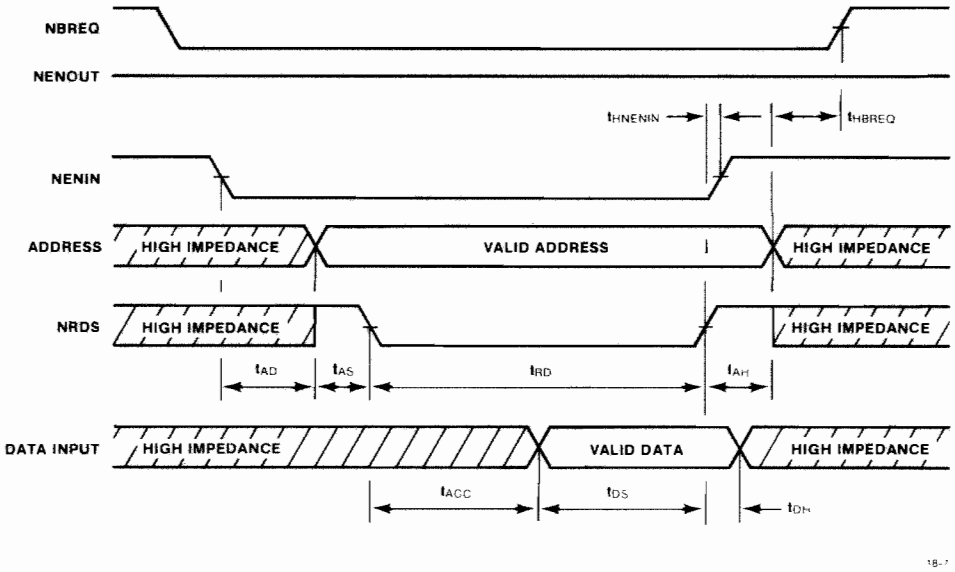
To initiate data transfers with external memory or peripherals, the 807X must gain access to the external address and data buses. The three pins associated with bus access control are NBREQ, NENIN, and NENOUT. For simple stand-alone applications, the NENOUT output pin may be ignored. The NENIN pin should be grounded or externally connected to NBREQ. The NBREQ pin must be connected to V_{CC} through an external pullup resistor.

2.2.2.6 Bus Access Control Logic

When the INS807X needs to use the external data bus, it must verify the availability of the bus. To accomplish this, the INS807X examines the state of the NBREQ pin. A logic low indicates that the data bus is busy. In this case, the INS807X sets the NENOUT pin to equal the state of the NENIN input and waits for the bus.

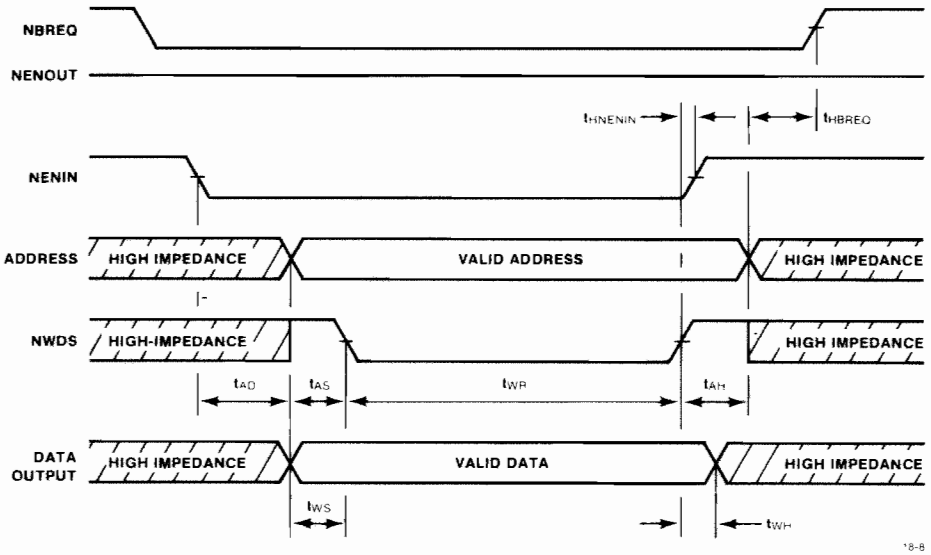
Bus availability is signalled by the NBREQ pin going high. The NBREQ line requires an external pull-up resistor to V_{CC} . After NBREQ goes high, the 807X issues a bus request by pulling NBREQ low. After this occurs, the 807X still cannot use the external buses until it is enabled (its active low enable input, NENIN, is brought to the low state).

After the 807X has succeeded in pulling NBREQ low, and has also received a low on its NENIN input, it will begin to execute an external READ or WRITE cycle. When the cycle is completed, the 807X releases the address bus, data bus and READ/WRITE strobes (these lines are set to the TRI-STATE mode) and it also ceases to hold NBREQ low.



READ CYCLE (fx = 4MHz max)

tx	=	$\frac{1}{fx}$
tRD	=	4tx ns typ
tAS	=	tx ns typ
tAH	=	tx ns typ
tACC	=	600
tDH	=	0
tAD	=	2tx ns typ
tHBREQ	=	50 typ
tHNENIN	=	0



WRITE CYCLE (fx = 4MHz max)

tWR	=	3tx ns typ
tAS	=	2tx ns typ
tAH	=	tx ns typ
tWS	=	tx ns typ
tWH	=	tx ns typ
tAD	=	2tx ns typ
tHBREQ	=	50 typ
tHNENIN	=	0

Figure 2 – 6. External Memory READ/WRITE Cycle Timing

When the 807X releases NBREQ, this line may or may not go high, depending upon whether it is being held low by other processors or DMA devices. If the NBREQ line goes high when released, the 807X leaves its NENOUT pin high and this completes its I/O Cycle. It the NBREQ does not go high when released, the 807X sets its NENOUT pin low at the end of its I/O cycle. Thus, the state of the NENOUT pin at the end of an I/O cycle depends upon whether or not the NBREQ line is allowed to go high after it has been released. This manipulation of the NENOUT pin makes it possible to implement 807X based DMA and multiprocessing systems using almost no external bus allocation hardware.

2.2.2.7 External READ/WRITE Cycle Extension

The NHOLD signal may be set low following the leading edge of the READ strobe (NRDS) or the WRITE strobe (NWDS). Setting NHOLD low causes the 807X to extend an external I/O (READ or WRITE) cycle. When an external I/O cycle is extended, the 807X address, data and strobe lines are held active until the NHOLD signal is returned high. There is no restriction on the maximum duration of NHOLD signal in the low state. Thus, NHOLD can be utilized in a variety of ways, ranging from accommodation of slow memories/peripherals to single READ/WRITE cycle execution for software debug purposes. Figure 2-7 compares I/O cycles with and without cycle extension.

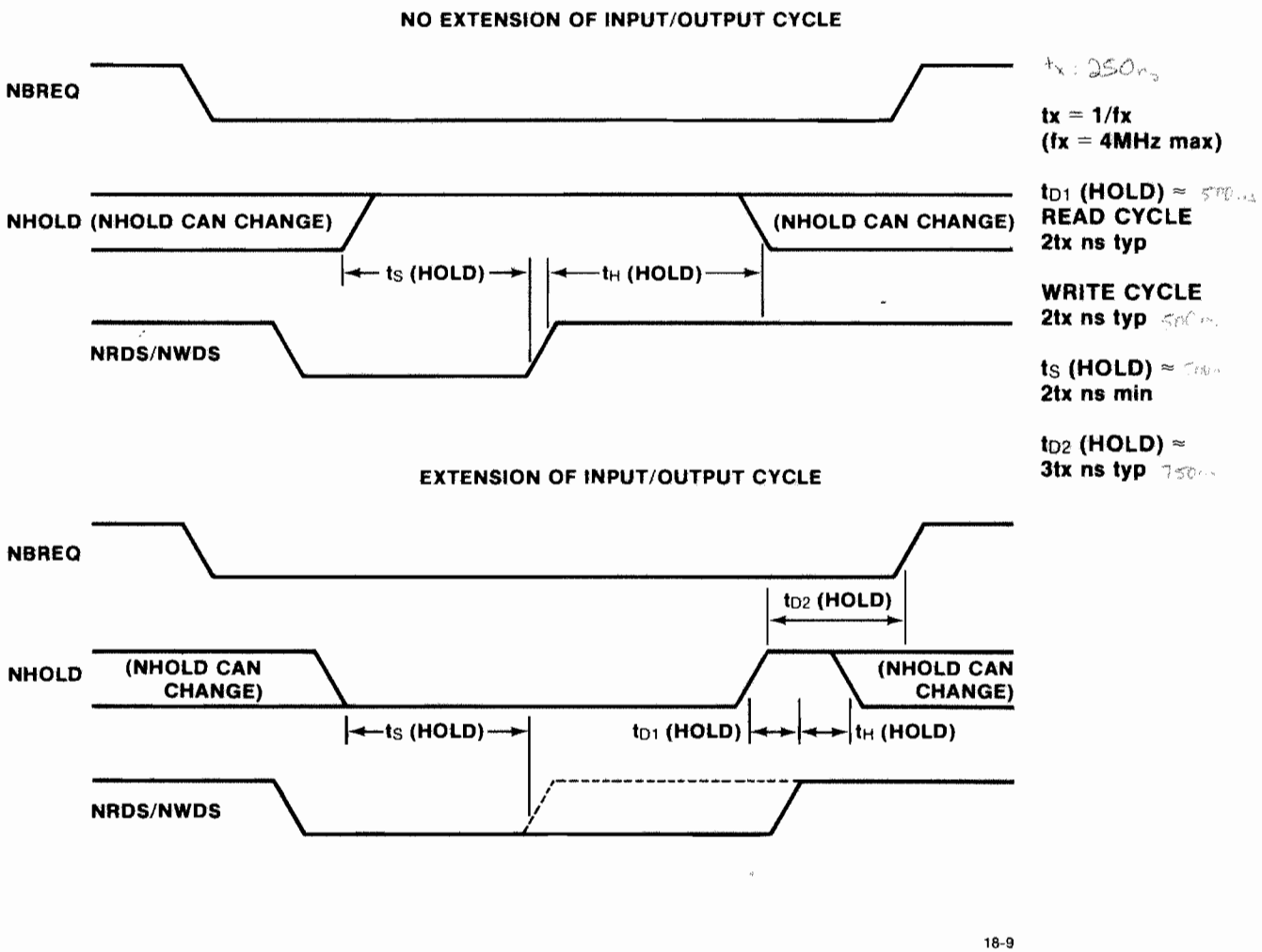


Figure 2 – 7. NHOLD Timing

2.2.2.8 Interrupt Servicing

When interrupts are enabled (IE = 1 in the STATUS register), pins SA/INTA and SB/INTB act as vectored interrupts. Both inputs are edge triggered, and are activated by the high-to-low logic transition. (Edge triggered interrupts offer the advantage of not requiring an interrupt acknowledge signal to remove the interrupt service request, as required by level triggered interrupts.) Interrupt service requests are latched when they occur, and are recognized prior to each instruction fetch. When an interrupt is recognized, the contents of the program counter are pushed onto the stack, saving the return address. The interrupt enable bit (IE) in the STATUS register is automatically reset, disabling further interrupts from being recognized, but interrupts occurring during this time will still be latched. The processor then fetches the next instruction from a pre-determined memory location. The location for interrupt A is 0004_{16}, and the location for interrupt B is 0007_{16}. (Refer to figure 2-8.) The instructions stored in these vector locations are normally BRANCH or JUMP instructions, causing the processor to begin execution of the appropriate service subroutine. The interrupt service subroutines may be of arbitrary length and may be located anywhere in memory.

Since the interrupt enable bit (IE) is automatically cleared at the start of interrupt service, Interrupt A cannot normally interrupt itself or Interrupt B, and vice versa, unless interrupts are re-enabled in the service subroutine. Since interrupt requests are latched, an interrupt request will never be lost, even though it occurs during servicing of the other interrupt.

Interrupt service subroutines should always be concluded with an interrupt enable instruction (OR S = 01_{16}) immediately followed by a return instruction (RET). This sequence causes the processor to resume main program execution at the point where it was interrupted.

MEMORY LOCATION	MEMORY CONTENTS	
0	—	
1	JMP	RESTART
2	XX	
3	XX	
4	JMP	INTERRUPT A
5	XX	
6	XX	
7	JMP	INTERRUPT B
8	XX	
9	XX	
A	—	

Figure 2-8. Interrupt Transfer Locations 18-10

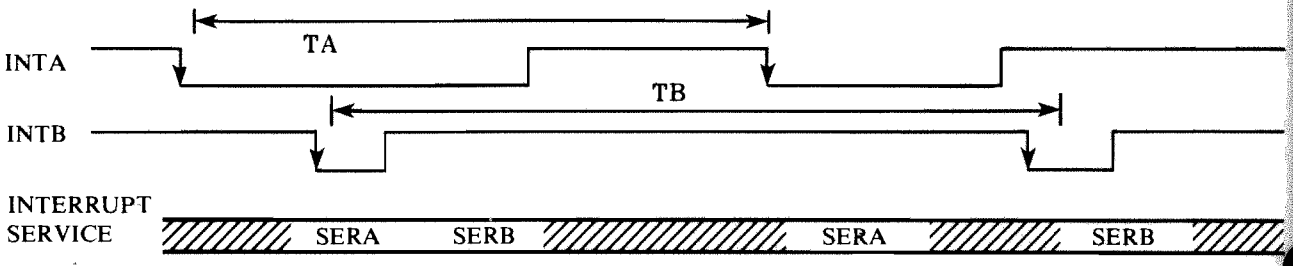
2.2.2.9 Interrupt Timing and Interrupt Priority

Interrupts are latched when they occur and are sampled prior to each instruction fetch. Thus, when an interrupt occurs during execution of an instruction, the instruction in progress must be completed before the interrupt will be recognized. This results in an interrupt recognize latency time (IRLT), that is a function of the time at which the asynchronous interrupt occurs. The minimum value for the IRLT is 9 microcycles, and the maximum value is equal to the duration of the longest program instruction (in microcycles) plus 9 microcycles.

If both interrupts occur during the IRLT, Interrupt A will always be recognized first (see figure 2-9), even though Interrupt B may have occurred slightly earlier. In order for Interrupt B to be recognized before Interrupt A, Interrupt B must occur prior to Interrupt A by an amount greater than the maximum value of the IRLT.

If several interrupt requests occur on the same pin during the IRLT, they will be treated as a single interrupt request. If two interrupt requests occur on the same pin spaced apart by a time duration greater than the IRLT, they will be treated as two separate requests. For this case, if the second interrupt occurs during interrupt service, service will not be interrupted, unless interrupts have been re-enabled by the interrupt service routine.

The interrupt service latency time (ISLT) is the time duration measured from the active edge of the interrupt request to the start of execution of the first instruction in the interrupt service routine. When interrupts are enabled, the ISLT is equal to the IRLT plus 8 microcycles or 5 microcycles, depending upon whether a JUMP or BRANCH instruction is stored at the vector memory locations 0004_{16} and 0007_{16}.



= Main program in progress

SERA = Interrupt A being serviced

SERB = Interrupt B being serviced

TA = Maximum frequency (minimum period) of interrupt A request

TB = Maximum frequency (minimum period) of interrupt B request

NOTE: If interrupt A and interrupt B are close to being "periodic" (not single event or burst mode) then for the main program to be efficiently executed the following conditions must be met:

$$TA >> SERA + SERB$$
$$TB >> SERA + SERB$$

Figure 2-9. Interrupt Service Timing

18-11

2.2.3 Instruction Set

The 807X instruction set consists of 192 instructions which can be grouped into nine general categories:

1) Register Load/Store Instructions.
2) Register to Register Exchange Instructions.
3) Arithmetic Operations: Add, Subtract, Multiply and Divide.
4) Boolean Operations: AND, OR, Exclusive OR.
5) Shift and Rotate Instructions.
6) Stack PUSH/POP Instructions.
7) Program Counter Modification Instructions.
8) Text Processing Instructions.
9) Miscellaneous Instructions.

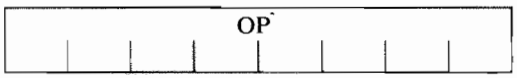
A listing of the complete instruction set is provided in Chapter 3.

When external memory is accessed, additional microcycles must be added because external memory I/O requires an additional microcycle per READ or WRITE operation. In addition to data READ and WRITE, external READ operations to fetch the op code and displacement (if required) must be taken into consideration.

The 807X's ability to perform operations on double-byte operands, as well as on single-byte operands is especially significant. This capability often results in higher computation accuracy, faster program execution, and smaller program size. Double-byte operands are fetched from memory one byte at a time. The least-significant operand byte is fetched first; the most-significant byte is fetched from the next higher memory location.

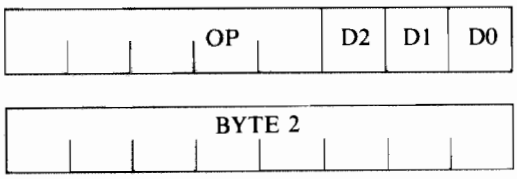
All 807X instructions fall into three categories: single, double, and triple byte. Figure 2-10 indicates the basic format for all three instruction categories.

One-Byte Instructions



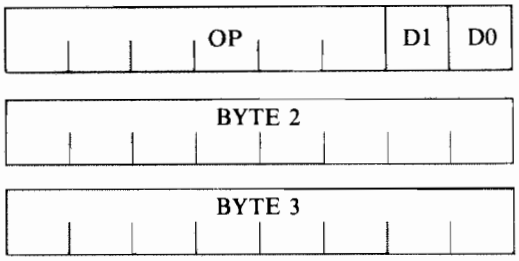
OP completely specifies the operation to be performed. No further memory references are required.

Two-Byte Instructions



OP specifies the operation to be performed. The addressing mode is determined by bits D0, D1 and D2. For all addressing modes except immediate, BYTE 2 represents a displacement value between $+127_{10}$ and -128_{10} which is used to calculate the effective memory address. When the immediate mode is specified, BYTE 2 contains data.

Three-Byte Instructions



If OP specifies an operation on which D0 and D1 specify a pointer (PC, SP, P2 or P3), bytes 2 and 3 represent a 16-bit address.

If OP specifies an operation which does not involve a pointer, bytes 2 and 3 represent 16 bits of immediate data. Byte 2 is the least significant byte, and Byte 3 is the most significant byte.

Figure 2 – 10. 807X Instruction Format

2.2.4 Addressing Modes

The 807X has addressing modes which can be grouped into nine basic categories:

1) Implied.
2) PC Relative.
3) Pointer Relative (via SP, P2 or P3).
4) Single-Byte Immediate and Double-Byte Immediate.
5) Direct.
6) Auto-Indexed (via P2 or P3).
7) Indexed (via P2 or P3).
8) Absolute.
9) Indirect.

2.2.4.1 Implied Addressing

Implied addressing is used for all single-byte instructions. For this instruction category, the operation to be performed is completely specified by the op code; no further memory references are required. The instruction types that utilize implied addressing include the following:

- Register-to-Register Load/Exchange.
- All Operations Between the E Register and the Accumulator.
- Multiply/Divide Instruction.
- Shifts and Rotates on the Accumulator.
- PUSH/POP/RETURN Instruction.

2.2.4.2 PC Relative Addressing

A PC relative address is formed by adding the displacement value specified in the second byte of the instruction to the current contents of the program counter. The displacement value is an 8-bit two's complement number, giving a relative addressing range of -128_{10} to $+127_{10}$ locations from the current contents of the program counter.

2.2.4.3 Pointer Relative Addressing

A pointer relative address is formed by adding the displacement value specified in the second byte of the instruction to the current value of a specified pointer. Pointer SP, P2 or P3 may be specified. (Refer to figure 2-11.)

Type of Addressing	OP CODE Bits D2	D1	D0	Pointer Specified	Displacement
PC Relative	0	0	0	PC	−128 to +127
Pointer Relative	0	0	1	SP	−128 to +127
	0	1	0	P2	−128 to +127
	0	1	1	P3	−128 to +127
Immediate	1	0	0	N/A	−128 to +127
Direct	1	0	1	N/A	00_{16} to FF_{16} See Note 1. below
Auto Indexed	1	1	0	P2	−128 to +127
	1	1	1	P3	−128 to +127

Note 1: The effective address is calculated by adding the displacement value (00_{16} to FF_{16}) to $FF00_{16}$. Displacement values of $C0_{16}$ to FF_{16} specify on-chip RAM (on-chip RAM is located from $FFC0_{16}$ to $FFFF_{16}$).

Figure 2-11. Addressing Mode Specification

The displacement value is an 8-bit two's complement number, giving a relative addressing range of -128_{10} to $+127_{10}$ locations for the contents of the pointer specified. The contents of the specified pointer are not altered.

2.2.4.4 Immediate Addressing

When single-byte immediate addressing is specified, the next byte following the op code is used as the operand. When double-byte immediate addressing is specified, the first byte following the op code is the least-significant byte of data and the second byte following the op code is the most-significant byte of data.

2.2.4.5 Direct Addressing

A direct address is formed by adding the 8-bit displacement specified in the second byte of the instruction to the direct address $FF00_{16}$. Thus, direct addressing can address 256 memory locations ranging sequentially from $FF00_{16}$ to $FFFF_{16}$. Since on-chip RAM is located from $FFC0_{16}$ to $FFFF_{16}$, displacement values of $C0_{16}$ to FF_{16} will address on-chip RAM. When the direct addressing mode is specified, on-chip RAM can be addressed without dedicating a pointer.

2.2.4.6 Auto-Indexed Addressing

An auto-indexed address is formed by adding the displacement value contained in the second byte of the instruction to the current value of a specified pointer. Pointers P2 or P3 may be specified. The displacement value is an 8-bit, two's complement number, giving a relative addressing range of -128_{10} to $+127_{10}$ locations from the contents of the pointer. The contents of the specified pointer are altered by auto-indexed addressing as follows: if the displacement is less than zero, it is added to the current contents of the pointer and the result is stored in the pointer. The contents of the pointer are then used as the effective address. If the displacement is greater than or equal to zero, the current contents of the pointer are used as the effective address. The pointer is then incremented by the displacement after the contents of the effective address are fetched or stored.

A major advantage of auto-indexed addressing is that it allows multiple software stacks to be created, using P2 or P3 as software stack pointers. Thus, a PUSH A (A = accumulator) instruction could be implemented via an auto-indexed STORE A instruction, and a -1 displacement, from P2 or P3. Similarly, a POP A instruction could be implemented via an auto-indexed Accumulator LOAD instruction, and a $+1$ displacement from P2 or P3. The same techniques can be used to implement the equivalent of those instructions which PUSH and POP double bytes, i.e., PUSH EA and POP EA. The equivalents of these instructions can be implemented via auto-indexed, double byte LOAD and STORE instructions with a displacement of $+2$ and -2.

2.2.4.7 Indexed Addressing

The indexed addressing mode is used by the text processing instruction "search and skip if character matched" (SSM).

2.2.4.8 Absolute Addressing

Absolute addressing is used by the "JUMP" (JMP) and Jump to Subroutine (JSR) instructions. The destination address is specified by two bytes following the op code. The byte immediately following the op code is the least-significant address byte.

NOTE: The JMP and JSR instructions must specify a memory location which is one less than the location where program execution is to continue. Since the addresses in JMP and JSR instructions are normally specified as labels, the assembler calculates the correct address and the programmer need not be concerned. If it becomes necessary for the program to actually calculate a jump address, the above should be taken into account.

2.2.4.9 Indirect Addressing

The indirect addressing mode is used by the one-byte CALL Instructions.

2.2.5 Specialized Instructions

2.2.5.1 Search and Skip if Character Matched (SSM) Instruction

The SSM instruction searches memory for a character which matches the one present in the accumulator using indexed addressing. The address where the search begins may be specified by pointers P2 or P3. The number of memory locations searched is 256. The search is conducted by using the specified pointer to indicate the address of the character to be tested. The pointer is automatically incremented by 1 after each test. If a match is found within the allowable 256 searches, the next two program bytes will be skipped. In this case the instruction is concluded with the specified pointer pointing to an address one greater than that for which the match occurred.

If a match is not found within the 256 searches, the next instruction in sequence will be executed. In this case the specified pointer will have been incremented by one less than the number of searches allowed (255).

The SSM instruction is extremely versatile and may be utilized in a number of ways. As an example, consider the task of searching a line buffer for the end of the line, denoted by the ASCII carriage return character ($0D_{16}$). If the pointer P2 or P3 is initially pointing to the beginning accumulator, the SSM instruction will find the end of the line. After executing SSM the pointer (P2 or P3) will point to the next memory location following the end of the line.

2.2.5.2 Branch If Not Digit (BND) Instruction

The two-byte BND instruction tests the ASCII character present in the accumulator to determine whether or not it represents the ASCII equivalent of a decimal digit (30_{16} - 39_{16}). If it does not, a PC relative branch is executed, using the second byte of the instruction as displacement. In this case, the contents of the accumulator are not altered. If the accumulator does contain the ASCII equivalent of a decimal digit, the accumulator contents are converted to the 8-bit binary equivalent of the decimal digit (00_{16} - 09_{16}).

The BND instruction is extremely versatile and may be employed in a number of ways. As an example, consider the problem of searching an ASCII line buffer to locate the next ASCII decimal digit (30_{16} - 39_{16}) and converting it to its BCD equivalent (00_{16} - 09_{16}). This operation could be part of the more complex task to locate and convert to BCD, a multiple digit number represented in ASCII. This task can be efficiently accomplished using only an auto-indexed load accumulator instruction and a BND instruction as follows:

```
LOOP: LD A, @ +1 (P2)
      BND LOOP

NEXT: (Next Instruction)
```

2.2.5.3 One-Byte Call Instructions

The CALL and JSR instructions are similar in that both allow the user to invoke a subroutine. A major difference, however, is that the CALL instruction occupies only one byte, whereas the JSR instruction occupies three bytes. The CALL instruction utilizes indirect addressing and allows the user to invoke up to 16 subroutines. Each of the 16 subroutines may be of arbitrary length and located anywhere in memory. The two-byte pointer address for each subroutine is stored in two consecutive memory locations, starting at 32_{10} (20_{16}). (Refer to Figure 2-12.) The values of the subroutine starting addresses stored in these indirect locations should be one less than the memory location where the subroutine actually begins. Subroutines invoked via a one-byte CALL are exited in the normal manner using a RET instruction.

2.2.5.4 PUSH and LOAD IMMEDIATE (PLI) Instruction

The three-byte PLI instruction pushes the designated pointer register's contents (P2 or P3) on the stack and then loads the pointer with the immediate address contained in the second and third bytes of the instruction. The contents of the designated pointer are stored on the stack with the least-significant pointer byte located at the lower memory address. The stack pointer is automatically incremented by two.

The PLI instruction is especially useful when a program requires many pointers to data, tables, I/O, etc. For this case, pointers P2 and P3 can be efficiently multiplexed among many tasks. This can be accomplished by simply executing a PLI instruction immediately before each task is to be executed, and a POP pointer instruction when each task is completed. This procedure causes the old pointer value to be saved on the stack, and the new pointer value to be loaded into the pointer. The given task is then executed using the pointer as required. When the task is concluded and the pointer contents are no longer needed, the old pointer value is restored into the pointer from the stack by executing a POP P2 (or P3) instruction.

MEMORY LOCATION (DECIMAL)	MEMORY LOCATION (HEX)	MEMORY CONTENTS	
00032	0020	ADDRL 0	
33	21	ADDRH 0	CALL 0
34	22	ADDRL 1	
35	23	ADDRH 1	CALL 1
36	24		
37	25		CALL 2
38	26		
39	27		CALL 3
40	28		
41	29		CALL 4
42	2A		
43	2B		CALL 5
44	2C		
45	2D		CALL 6
45	2E		
47	2F		CALL 7
48	30		
49	31		CALL 8
50	32		
51	33		CALL 9
52	34		
53	35		CALL 10
54	36		
55	37		CALL 11
56	38		
57	39		CALL 12
58	3A		
59	3B		CALL 13
60	3C		
61	3D		CALL 14
62	3E	ADDRL 15	
00063	003F	ADDRH 15	CALL 15

NOTE: ADDRL designates the lower byte of the starting address.
ADDRH designates the upper byte of the starting address.

Figure 2-12. Memory Locations for the Destination Addresses for CALL.

2.2.5.5 Increment and Load (ILD), Decrement and Load (DLD) Instructions

The ILD instruction reads the contents of the effective address, increments the contents by 1, and stores the 8-bit result in the effective address and in the accumulator. Similarly, the DLD instruction reads the contents of the effective address, decrements the contents by 1, and stores the 8-bit result in the effective address and in the accumulator.

Thus, the ILD/DLD instructions basically perform a READ/MODIFY/WRITE operation on the contents of a memory location. In multiprocessing applications, several processors may perform an ILD or DLD operation on the same flag (memory location) in order to determine the next task to be performed and/or which processor will execute it. In order to insure correct task assignment, the ILD/DLD instructions retain the bus between the time that the initial contents of the effective address are fetched and the modified data are stored.

2.2.5.6 Multiply Instruction

The 807X multiplication instruction (MULT) multiplies a signed 16-bit multiplicand by a signed 16-bit multplier, generating a signed 32-bit product. The signed 16-bit multiplicand may be positive or negative and is stored in the double-byte accumulator EA. (EA = the concatenated register E and A). The signed 16-bit multiplier must be positive and is storèd in the 16-bit register T. The signed 32-bit product is stored in the EA and T registers, replacing the original multiplier and multiplicand. The most-significant 16-bits of the product are stored in the EA register, and the least-significant 16-bits of the product are stored in the T register. In shorthand notation, the multiplication instruction may be written as follows:

$$(EA) \leftarrow [(EA) * (T)]_{31:16}$$

$$(T) \leftarrow [(EA) * (T)]_{15:0}$$

NOTE: The subscripts represent bit positions of the product.

The multiplication instruction is executed in only 37 microcycles (37 μsec) when the processor is operating at an oscillator frequency of 4 MHz.

2.2.5.7 Divide Instruction

The 807X divide instruction divides a 16-bit dividend by a 16-bit divisor, generating a 16-bit quotient. Since the divide instruction is an integer divide, the magnitude of the dividend must be greater than the magnitude of the divisor, which must be non-zero. The 16-bit dividend can be an unsigned number in the range 0_{10} - $65{,}535_{10}$, or a signed positive number in the range 0_{10} - $32{,}767_{10}$. The divisor can be an unsigned number in the range 0_{10} - $32{,}767_{10}$, or a signed, positive number in the range 0_{10} - $32{,}767_{10}$. Thus, the divide instruction will divide a signed dividend by a signed divisor, when both quantities are positive. It will also divide an unsigned dividend in the range 0_{10} - $65{,}535_{10}$ by an unsigned divisor in the range 0_{10} - $32{,}767_{10}$.

Before division can be performed, a 16-bit dividend must be stored in the EA register and a 16-bit divisor must be stored in the T register. After division has been executed, a 16-bit quotient is stored in the EA register, replacing the original dividend; the divisor which has been stored in the T register is lost.

The division instruction is executed in only 42 microcycles (42μsec) when the processor is operating at an oscillator frequency of 4 MHz.

2.2.6 70-Series System Configuration

2.2.6.1 Interfacing with External Peripherals

Data are transferred on the 807X external data bus during execution of external memory reference instructions. This enables the 807X to use its entire memory reference class of instructions, including arithmetic and logic, to implement input/output (I/O) to external peripherals. Since all memory reference instructions are available for peripheral I/O, more efficient service routines can be written which execute faster and require fewer memory bytes.

The 807X can implement serial I/O very efficiently via its two sense inputs (SA/INTA, SB/INTB) and three flag outputs (Fi, F2, F3). Using these, two independent serial I/O channels can be readily implemented under software control using one sense input and one flag output per channel. Devices which can use this feature include Teletypes, terminals, and modems.

2.2.6.2 Bus Access Control in Multiprocessing Systems

A typical 807X based multiprocessing system contains several 807X processors which share common memory and/or peripherals. This is accomplished by connecting together the corresponding address and data lines and READ and WRITE strobe outputs from each processor to form a "bus," and routing them to shared memory/peripherals as shown in figure 2-13.

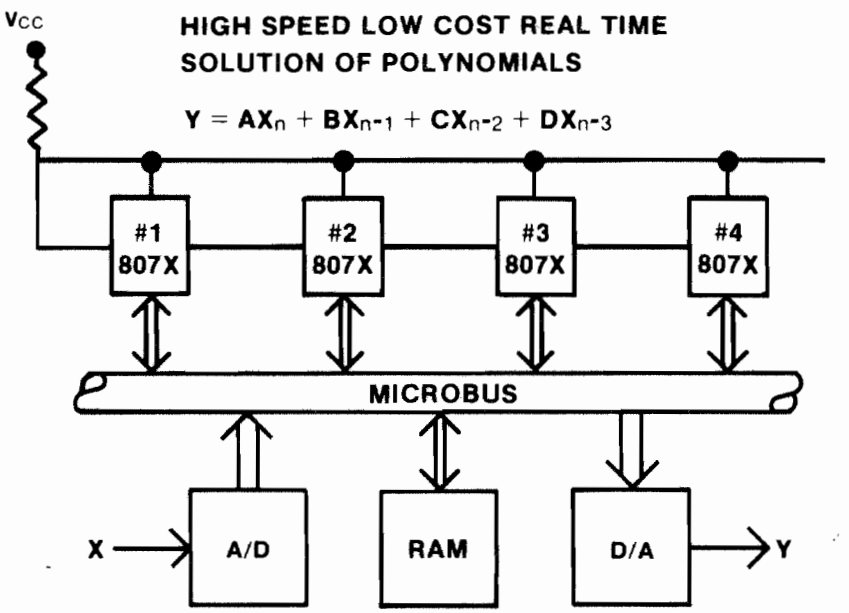


Figure 2 – 13. 807X Multiprocessing System 18-12

There are many methods of establishing priority between 807X processors in a multiprocessing system. One of the simplest approaches, which requires minimal external hardware, is illustrated in figure 2-13. The NENOUT pin from each 807X is connected to the NENIN pin of the following 807X, forming a "daisy chain." The NBREQ pins from all processors are connected to the NBREQ line which is pulled up to V_{CC} via a pullup resistor. This resistor is the only external hardware required for bus access control.

Whenever a processor must execute a READ or WRITE cycle to shared external memory, it must issue a bus request. The request is issued by pulling the NBREQ line low, only after it has gone high. When a processor succeeds in pulling NBREQ low, it also drives its NENOUT pin high. This high level ripples down the NENIN/NENOUT daisy chain, preventing all processors located further down the chain from executing an I/O cycle.

It is possible for several processors to issue a bus request concurrently by simultaneously pulling NBREQ low. When this occurs, those processors that have pulled NBREQ low will be granted use of the bus before those processors that have not. Thus, the processors that have pulled NBREQ low will have higher bus priority than those that have not done so. This higher priority results from the fact that the NBREQ line will not be allowed to go high until all processors that have NBREQ pulled low have been granted use of the bus.

Priority among those 807X processors that have simultaneously pulled NBREQ low is established by the on-chip bus allocation logic. This logic assures that the closer a given processor is to the beginning of the NENIN/NENOUT daisy chain, the higher its bus priority. It is important to note, however, that this only applies to those processors that are simultaneously holding NBREQ low.

Note that processor #1, the first processor in the NENIN/NENOUT daisy chain, has its NENIN pin connected to the NBREQ line. Thus, whenever this processor succeeds in pulling NBREQ low, it will always be granted use of the bus before all other processors. Whenever processor #1 succeeds in pulling NBREQ low, it always has the highest bus priority. Those processors that are holding NBREQ low pass usage of the bus from one to another, as follows. During normal operation, a processor can retain the bus for 2 microcycles, which is long enough to execute a single READ or WRITE cycle. (1 microcycle = μsec when the INS807X is running at the 4 MHz maximum frequency.) When a given processor is finished using the bus, it TRI-STATES its address, data and strobe pins. It also ceases to hold NBREQ low. When other processors are concurrently holding NBREQ low, NBREQ will not rise. In this case, the processor that is releasing the bus will drive its NENOUT pin low. Since this pin is connected to the NENIN pin of the next processor, the latter is enabled to execute on external I/O cycle only if it is holding NBREQ low. If not, this processor will pass the level on its NENIN pin to its NENOUT pin. Thus, the active low enable signal will propagate down the NENIN/NENOUT daisy chain until it reaches a processor that is holding NBREQ low.

When this occurs, the processor holding NBREQ low is enabled to execute an external I/O cycle. When the cycle is concluded, the sequence described above is repeated. Thus, the active low enable signal propagates from the NENIN/NENOUT daisy chain, enabling one processor at a time, until all processors holding NBREQ low have been granted use of the bus. When the last processor holding NBREQ low ceases to do so, the NBREQ line will rise. At this time, all processors that are waiting to issue a bus request may do so by pulling NBREQ low. In this manner, all processors may request and be granted use of the bus, no matter where they are located in the NENIN/NENOUT daisy chain. No processor will ever be "locked out," even though it may have a low bus priority.

In most INS807X multiprocessing applications, each processor generally spends most of its time executing instructions from internal memory. Since internal memory operations do not use the external bus, many INS807X processors can share the same bus. This often results in high system throughput, which can be almost directly proportional to the number of processors on the bus.

Although the 807X on-chip bus allocation logic can function as described above, it is still compatible with most types of external bus control schemes. The NBREQ output from each 807X generally serves to indicate when use of the external bus is required. The NENIN input to each 807X may be used to enable the execution of an external READ or WRITE cycle.

2.2.6.3 Use of the INS807X in DMA Systems

The 807X pins NENIN, NENOUT, and NBREQ make it possible to implement DMA systems with minimal external hardware. Figure 2-14 shows how a typical DMA system might be organized. The system contains an INS807X processor, a DMA unit, and shared memory (and/or I/O). The external hardware required to implement DMA consists of only a single quad NAND package plus four resistors. When the memory is not being used by the 807X or the DMA unit, the address, data and strobes lines are in the high impedance state. Resistors connected to the READ and WRITE strobe lines insure that these signals remain inactive when the 807X and DMA unit are not using the bus. The DMA logic shown in figure 2-14 functions as follows. The 807X or the DMA unit can request use of the bus by pulling NBREQ low, *only if it is presently high.* If both units *simultaneously* pull NBREQ low, the DMA unit has priority and will be serviced first. When the DMA unit is finished using the bus, it sets DMA Request low. This causes the DMA Acknowledge signal to go low, enabling the INS807X to use the bus by setting NENIN low and NBREQ high.

The simple DMA logic can be easily extended to more complex DMA systems containing several DMA units and several 807X processors, all sharing common memory (or I/O) on the same bus.

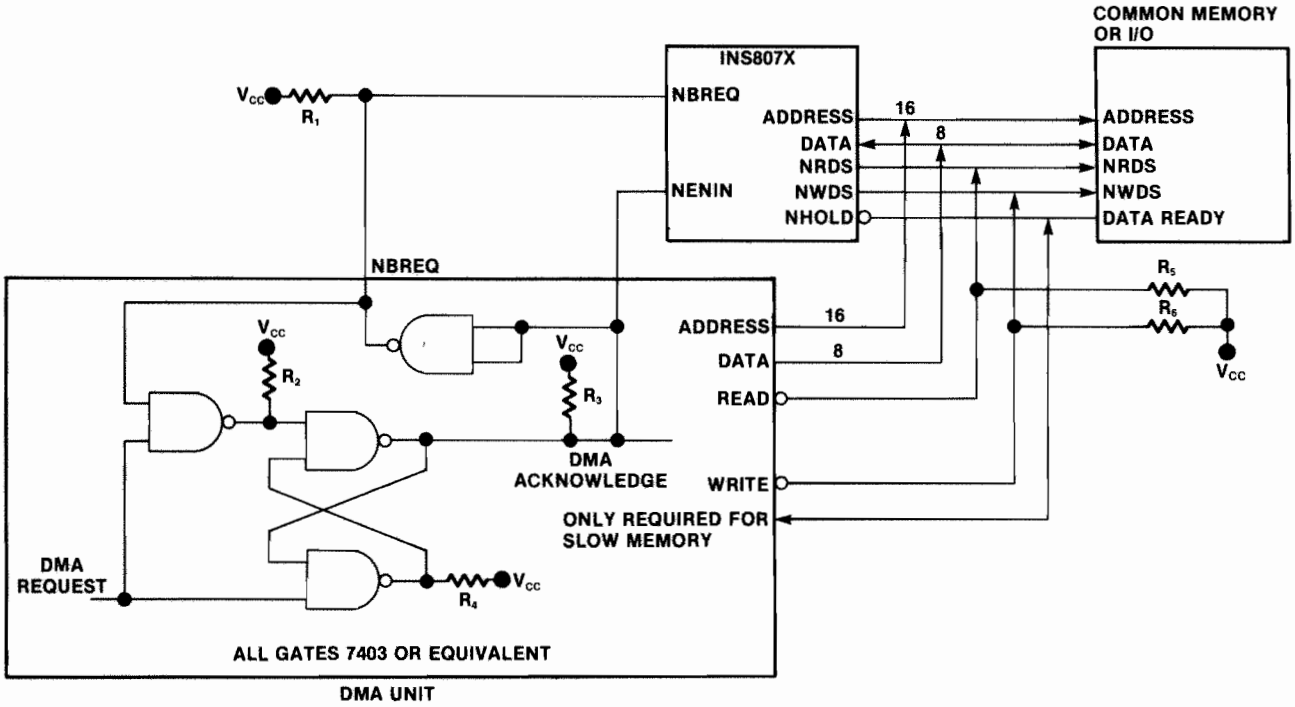


Figure 2 – 14. INS807X in a Typical DMA System

18-13

Chapter 3

INSTRUCTION SET

The assembly language instruction set of the 70-Series provides arithmetic, logic, shift, transfer, and other operations among the accumulator, the other registers, and memory.

Instruction statements, when assembled, generate the object (machine) code that defines the operations the processor will perform. Depending on the instruction type, one or two bytes of object code are generated for each instruction assembled.

In the following descriptions, any user-accessible register or bit that is not explicitly mentioned will not be altered by the instruction. The diagram with each instruction shows the bit configuration for the specified instruction. Low order bits are always at the right.

There are 192 instruction statements executable by the 70-Series family.

Refer to table 3-1 for definitions of the symbols used in the notation for describing the 807X instruction set. Upper-case mnemonics refer to units designated by fields of the instruction words; lower-case mnemonics refer to the numerical values of the corresponding fields. For example, ptr in an assembler statement denotes the number of a pointer register, whereas (AC) ← (PTR) denotes the contents of the accumulator are replaced by the contents of a pointer register. In the case where both a lower-case mnemonic and an upper-case mnemonic are composed of the same letters, only the lower-case mnemonic is given in table 3-1. Lower-case notation designates a variable.

The INS807X instruction set is summarized in appendix A.
The OP code listing is presented in appendix B.

Refer to chapter 5 for a full description of other statements available in the 807X assembler.

Table 3-1. Symbols and Notation

Symbol and Notation	Meaning
A	8-bit accumulator.
address	In the operand field of memory-reference, memory increment/decrement or transfer instruction source statements, address indicates the symbolic expression that represents a memory location. The assembler calculates the displacement (disp) for the instruction from address and generates the instruction with PC-relative addressing.
CY/L	Carry/link flag in the status register.
data	8-bit immediate data field. Data may represent a signed or an unsigned two's complement number or a binary coded decimal (BCD) number.
disp	Displacement represents a signed 8-bit address modifier in a memory-reference, memory increment/decrement or transfer instructions.
EA	Extended accumulator. Concatenation of the extension register and the accumulator register.
E	Extension register provides for temporary storage, variable displacements and separate serial input/output port.
i	Represents the bit in one of the bit positions, 7 through 1, of the accumulator or the extension register.
IF	Interrupt enable flag.

Table 3-1. Symbols and Notation (Continued)

Symbol and Notation	Meaning
m	Mode bit, used in memory reference instructions. Blank parameter sets m = 0, @ sets m = 1.
OV	Overflow flag in the status register.
PC	Program counter (Pointer Register 0); during address formation, PC points to the last byte of the instruction being executed.
ptr	Pointer register (ptr = 0 through 3). The register specified in byte 1 of the instruction.
$ptr_{n:m}$	Pointer register bits; n: m = 7 through 0 to 15 through 8.
SP	Stack pointer.
S	8-bit status register.
r	Designates PC, SP, P2 or P2. Refer to Figure 2-11 for bit configuration of the selected register.
( )	Means "contents of." For example, (EA) is contents of extended accumulator.
[ ]	Means optional field in the assembler instruction format.
~	Ones complement of value immediately to right of ~.
→	"Replaces".
←	"Is replaced by".
↔	"Exchange".
*	Multiplication.
@	When used in the operand field of the instruction, sets the mode bit (m) to 1 for auto-incrementing/auto-decrementing indexing.
∧	AND operation.
∨	Inclusive-OR operation.
⊻	Exclusive-OR operation.
≥	Greater than or equal to
≤	Less than or equal to
=	Equals.
≠	Does not equal.

3.1 ADD EXTENSION TO ACCUMULATOR

Formats:

ADD A,E ;Implied

0	1	1	1	0	0	0	0

Operation: (A) ← (A) + (E)

The contents of the extension register are added to the contents of the accumulator. The result is stored in the accumulator.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	4
Addressing:	Implied
Flags Affected:	CY/L, OV

Example:

ADD A,E ;Implied

3.2 ADD IMMEDIATE TO ACCUMULATOR

Formats:

ADD A, =data ;Immediate

1	1	1	1	0	1	0	0
data							

Operation: (A) ← (A) + **data**

The data in byte 2 of the instruction are added to the contents of the accumulator. The result is stored in the accumulator.

Data must be specified as a literal.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY; L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	5
Addressing:	Immediate
Flags Affected:	CY/L, OV

Examples:

ADD A, = VALUE ; Immediate
ADD A, = 0FF ; Immediate

3.3 ADD IMMEDIATE TO EXTENDED ACCUMULATOR

Formats:

```
ADD     EA, = data                    ;Immediate
```

1	0	1	1	0	1	0	0
data (low byte)							
data (high byte)							

Operation: (EA) ← (EA) + data + 1, data

The data in byte 3 and byte 2 of the instruction are added to the contents of the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Data must be specified as a literal. Referencing a label as a literal value will reference the address assigned to the label. CY/L is set if a carry from bit 7 of the extension register occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	8
Addressing:	Immediate, double precision
Flags Affected:	CY/L, OV

Examples:

```
ADD     EA, = 0FF24                   ;Immediate
ADD     EA, = DATA                    ;Immediate, references address of DATA
```

3.4 ADD MEMORY TO ACCUMULATOR

Formats:

```
        A,label                       ;PC relative, PC is assumed
ADD     A,disp,r                      ;PC or PTR relative, r = PC, SP, P2, or P3
        A,@disp,r                     ;Auto-indexed, r = P2 or P3
        A,disp,@r                     ;Auto-indexed, r = P2 or P3
```

1	1	1	1	0	@	r
disp						

Operation: (A) ← (A) + (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" are added to the contents of the accumulator. The result is stored in the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	7, 9 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	CY/L, OV

Examples:

```
ADD     A,TEMP          ;PC relative
ADD     A,TEN,PC        ;PC relative
ADD     A,-35,P2        ;P2 relative
ADD     A,TEN,P3        ;P3 relative
ADD     A,TEMP,@P2      ;Auto-indexed
ADD     A,@2,P3         ;Auto-indexed
```

3.5 ADD MEMORY TO EXTENDED ACCUMULATOR

Formats:

```
ADD     EA,label        ;PC relative, PC is assumed
ADD     EA,disp,r       ;PC or PTR relative, r = PC, SP, P2, or P3
ADD     EA,@disp,r      ;Auto-indexed, r = P2 or P3
ADD     EA,disp,@r      ;Auto-indexed, r = P2 or P3
```

1	0	1	1	0	@	r
disp						

Operation: (EA) ← (EA) + (addr + 1, addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" are added to the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

CY/L is set if a carry from bit 7 of the extension register occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	10, 12 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	CY/L, OV

Examples:

```
ADD     EA,DATA         ;PC relative
ADD     EA,02F,PC       ;PC relative
ADD     EA,TEMP,P3      ;P3 relative
ADD     EA,@2,P3        ;Auto-indexed by P3
ADD     EA,2,@P2        ;Auto-indexed by P2
```

3.6 ADD MEMORY DIRECT TO ACCUMULATOR

Formats:

ADD A,label ;Direct 0FF00 ≤ label ≤ 0FFFF

1	1	1	1	0	1	0	1
disp							

Operation: (A) ← (A) + (addr); addr = 0FF00 + disp

The contents of memory location "addr" are added to the contents of the accumulator. The result is stored in the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must be in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles:	7
Addressing:	Direct
Flags Affected:	CY/L, OV

Examples:

ADD	A,TEMP	;Direct
ADD	A,0FFC5	;Direct

3.7 ADD MEMORY DIRECT TO EXTENDED ACCUMULATOR

Formats:

ADD EA, label ;Direct 0FF00 ≤ label ≤ 0FFF

1	0	1	1	0	1	0	1
disp							

Operation: (EA) ← (EA) + (addr + 1, addr); addr = 0FF00 + disp

The contents of memory locations "addr + 1" and "addr" are added to the contents of the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Label may be specified as a constant, symbol, or expression. Its value must be in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

CY/L is set if a carry from bit 7 of the extension register occurred; otherwise CY/L is cleared. OV is set if the sign of the result differs from the sign of both operands; otherwise OV is cleared.

Cycles: 10
Addressing: Direct, double precisions
Flags Affected: CY/L, OV

Example:

```
ADD     EA,DUMMY+1          ;Direct, double precision
```

3.8 AND EXTENSION WITH ACCUMULATOR

Formats:

```
AND     A,E                 ;Implied
```

0	1	0	1	0	0	0	0

Operation: (A) ← (A) ∧ (E)

The contents of the extension register are ANDed with the contents of the accumulator. The result is stored in the accumulator.

The truth table for this instruction is shown below.

A b	E b	A b ∧E b
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7 through 0

Cycles: 4
Addressing: Implied
Flags Affected: None

Example:

```
AND     A,E                 ;Implied
```

3.9 AND IMMEDIATE WITH ACCUMULATOR

Formats:

```
AND     A, =data            ;Immediate
```

1	1	0	1	0	1	0	0
data							

Operation: (A) ← (A) data

The data in byte 2 of the instruction are ANDed with the contents of the accumulator. The result is in the accumulator.

Data must be specified as a literal.

The truth table for this instruction is shown below.

A b	data b	A b ∧ data b
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7 through 0

Cycles: 5
Addressing: Immediate
Flags Affected: None

Example:

```
AND     A, = -3          ;Immediate
```

3.10 AND IMMEDIATE WITH STATUS REGISTER

Formats:

```
AND     S, = data        ;Immediate
```

0	0	1	1	1	0	0	1
data							

Operation: (S) ← (S) ∧ data

The data in byte 2 of the instruction are ANDed with the contents of the status register. The result is stored in the status register.

Data must be specified as a literal.

The truth table for this instruction is shown below.

S b	data b	S b ∧data b
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7, 6, 3, 2, 1, 0

Cycles: 5
Addressing: Immediate
Flags Affected: CY/L, OV, F3, F2, F1, IE

Example:

```
AND     S, = 0E0         ;Immediate
```

3.11 AND MEMORY WITH ACCUMULATOR

Formats:

```
AND    A,label        ;PC relative, PC is assumed
AND    A,disp,r       ;PC or PTR relative, r = PC, SP, P2, or P3
AND    A,@disp,r      ;Auto-indexed, r = P2 or P3
AND    A,disp,@r      ;Auto-indexed, r = P2 or P3
```

1	1	0	1	0	@	r
disp						

Operation: (A) ← [A] ∧ (addr); addr = (r) + disp
If auto-index: (r) = (r) + disp

The contents of memory location "addr" are ANDed with the contents of the accumulator. The result is stored in the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

The truth table for this instruction is shown below.

A b	addr b	A b ∧ addr b
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7 through 0

Cycles: 7, 9 if auto-indexed
Addressing: PC relative, pointer relative, auto-indexed
Flags Affected: None

Examples:

```
AND    A,DATA         ;PC relative
AND    A,OFFSET,PC    ;PC relative
AND    A,0F0,SP       ;SP relative
AND    A,@0F,P2       ;Auto-indexed by P2
AND    A,-20,@P3      ;Auto-indexed by P3
```

3.12 AND MEMORY DIRECT WITH ACCUMULATOR

Formats:

```
AND     A,label                 ;Direct 0FF00 ≤ label ≤ 0FFFF
```

1	1	0	1	0	1	0	1
disp							

Operation: (A) ← A ∧ (addr); addr = 0FF00 + disp

The contents of memory location "addr" are ANDed with the contents of the accumulator. The result is stored in the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

The truth table for this instruction is shown below.

A b	addr b	A b ∧ addr b
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7 through 0

Cycles:	7
Addressing:	Direct
Flags Affected:	None

Example:

```
AND     A,BUFFER                ;Direct
```

3.13 BRANCH IF ACCUMULATOR NOT ZERO

Formats:

```
BNZ     label                   ;PC relative, PC is assumed

BNZ     disp,r                  ;PTR relative,
                                r = PC, P2 or P3
```

0	1	1	1	1	1	r
disp						

Operation: If (A) ≠ 0: (PC) ← addr; addr = (r) + disp

If the contents of the accumulator are not zero (any accumulator bit = 1), then program execution continues at the memory location specified by "addr" + 1.

If a label is specified, disp will be adjusted for pre-increment of PC and assembler will default to PC-relative addressing. If disp is specified, it will be assembled as is.

Disp may be specified as a constant, symbol, or expression.

Register r is specified as PC, P2, or P3. If a register is not specified, the assembler defaults to PC.

Cycles:	5
Addressing:	PC relative, pointer relative

Examples:

```
BNZ     START          ;PC relative
BNZ     0F, PC         ;PC relative
BNZ     START+2,P2     ;Pointer 2 relative
BNZ     0,P3           ;Pointer 3 relative
```

3.14 BRANCH IF ACCUMULATOR POSITIVE

Formats:

```
BP      label          ;PC relative, PC is assumed

BP      disp,r         ;PTR relative,
                       r = PC, P2, or P3
```

0	1	1	0	0	1	r
disp						

Operation: If (A) ≥ 0: (PC) ← addr; addr = (r) + disp

If the contents of the accumulator are positive or zero (accumulator bit 7 = 0), then program execution continues at the memory location specified by "addr" + 1.

If a label is specified, disp will be adjusted for pre-increment of PC and assembler will default to PC-relative addressing. If disp is specified, it will be assembled as is.

Disp may be specified as a constant, symbol, or expression.

Register r is specified as PC, P2, or P3.

Cycles:	5
Addressing:	PC relative, pointer relative

Examples:

```
BP      -15,PC         ;PC relative
BP      FINIS          ;PC relative
BP      FIVE,P2        ;Pointer 2 relative
BP      -15,P3         ;Pointer 3 relative
```

3.15 BRANCH IF ACCUMULATOR ZERO

Formats:

BZ	label	;PC relative, PC is assumed
BZ	disp,r	;PTR relative, r = PC, P2, or P3

0	1	1	0	1	1	r
disp						

Operation: If (A) = 0 : (PC) ← addr; addr = (r) + disp

If the contents of the accumulator are zero (accumulator bits 7 − 0 = 0), then program execution continues at the memory location specified by 'addr'' + 1.

If a label is specified, disp will be adjusted for pre-increment of PC and assembler will default to PC-relative addressing. If disp is specified, it will be assembled as is.

Disp may be specified as a constant, symbol, or expression.

Register r is specified as PC, P2, or P3.

Cycles:	5
Addressing:	PC relative, pointer relative

Examples:

BZ	NEXT	;PC relative
BZ	10,PC	;PC relative
BZ	−5,P2	;Pointer 2 relative
BZ	CTR,P3	;Pointer 3 relative

3.16 BRANCH IF NOT DIGIT IN ACCUMULATOR

Formats:

BND	label	;PC relative, PC is assumed
BND	disp,PC	;PC relative

0	0	1	0	1	1	0	1
disp							

Operation: If (A) is an ASCII digit: (A) ← (A) −030
If (A) is not an ASCII digit: (PC) ← addr; addr = PC + disp

If the contents of the accumulator are not an ASCII-coded decimal number (hexadecimal values 30 through 39), then program execution continues at the memory location specified by "addr'' + 1. The contents of the accumulator are not altered.

If the contents of the accumulator are an ASCII-coded decimal number, then the character is converted to an 8-bit binary number which replaces the ASCII number. In this case, execution will continue with the next instruction in sequence.

If a label is specified, disp will be adjusted for pre-increment of PC and assembler will default to PC-relative addressing. If disp is specified, it will be assembled as is.

Disp may be specified as a constant, symbol, or expression.

Cycles:	**9 if digit, 9 if not digit and**
Addressing:	**(A)> 039_{16}, 7 if not digit and (A) < 030_{16}.**

Example:

BND	5,PC	;Branch if not digit

3.17 BRANCH UNCONDITIONAL

Formats:

BRA	label	;PC relative, PC is assumed
BRA	disp,r	;PTR relative, r = PC, P2, or P3

0	1	1	1	0	1	r
disp						

Operation: (PC) ← addr; addr = (r) + disp

Program execution continues at the memory location specified by "addr" + 1.

If a label is specified, disp will be adjusted for pre-increment of PC and assembler will default to PC-relative addressing. If disp is specified, it will be assembled as is.

Disp may be specified as a constant, symbol, or expression.

Register r is specified as PC, P2, or P3.

Cycles:	5
Addressing:	PC relative, pointer relative
Flags Affected:	None

Examples:

BRA	20,PC	;PC relative
BRA	PROG	;PC relative
BRA	START + 5,P2	;Pointer 2 relative
BRA	3,P3	;Pointer 3 relative

3.18 CALL

Formats:

CALL data ;Indirect $0 \leq$ data ≤ 15

0	0	0	1	data

Operation: ((SP) - 1) ← (PCH) ← (021 + 2* data);
((SP) - 2) ← (PCL) ← (020 + 2* data);
(SP) ← (SP) - 2

The contents of the high-order 8 bits of the program counter are loaded into the memory location whose address is specified by the contents of the stack pointer - 1. Then the contents of memory location specified by "021 + 2* data" are loaded into the high-order bits of the program counter. The contents of the low-order 8 bits of the program counter are loaded into the memory location whose address is specified by the contents of the stack pointer - 2. Then the contents of the memory location specified by "020 + 2* data" are loaded into the low-order 8 bits of the program counter. The contents of the stack pointer are decremented by 2.

The CALL instruction allows the user to invoke up to 16 subroutines, designated as 0-15, as follows: CALL 0, CALL 1, CALL 2...CALL 15. Each of the 16 subroutines may be of arbitrary length and located anywhere in memory. The two-byte starting address for each subroutine is stored in two consecutive internal memory locations in ROM (see figure 3-1). The value of a subroutine starting address should be one less than the memory location where the subroutine actually begins. Subroutines invoked via a one-byte CALL are exited in the normal way using a RET instruction.

Cycles:	16
Addressing:	Indirect
Flags Affected:	None

Examples:

```
CALL    0       ;Indirect
CALL    SUB10   ;Indirect
```

MEMORY LOCATION (DECIMAL)	MEMORY LOCATION (HEX)	MEMORY CONTENTS	
00032	0020	ADDRL 0	CALL 0
33	21	ADDRH 0	
34	22	ADDRL 1	CALL 1
35	23	ADDRH 1	
36	24		CALL 2
37	25		
38	26		CALL 3
39	27		
40	28		CALL 4
41	29		
42	2A		CALL 5
43	2B		
44	2C		CALL 6
45	2D		
46	2E		CALL 7
47	2F		
48	30		CALL 8
49	31		
50	32		CALL 9
51	33		
52	34		CALL 10
53	35		
54	36		CALL 11
55	37		
56	38		CALL 12
57	39		
58	3A		CALL 13
59	3B		
60	3C	ADDRL 14	CALL 14
61	3D	ADDRH 14	
62	3E	ADDRL 15	CALL 15
00063	003F	ADDRH 15	

NOTE: ADDRL designates the lower byte of the starting address.
ADDRH designates the upper byte of the starting address.

Figure 3-1. Memory Locations of One-Byte Call Starting Addresses

3.19 DECREMENT AND LOAD ACCUMULATOR

Formats:

```
        A,label          ;PC relative, PC is assumed
DLD     A,disp,r         ;PC or PTR relative,
                         r = PC, SP, P2, or P3
        A,@disp,r        ;Auto-indexed, r = P2 or P3
        A,disp,@r        ;Auto-indexed, r = P2 or P3
```

1	0	0	1	1	@	r
disp						

Operation: (A), (addr) ← (addr) + 1, addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of the memory location specified by addr are decremented by 1 and then the result is loaded into the accumulator and addr. Auto-indexing, if specified, is then performed.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	8, 10 if auto-indexed
Addressing:	PC relative, PTR relative, auto-indexed
Flags Affected:	None

Example:

```
OLD     A,0F,PC          ;PC relative
```

Note: This is an interlocked instruction. If the effective address points to external memory, the processor will not release the bus between the read and write cycles. This allows effective implementation of semaphores.

3.20 DECREMENT AND LOAD ACCUMULATOR FROM MEMORY DIRECT

Formats:

```
DLD     A,label          ; Implied 0FF00 ≤ label ≤ 0FFFF
```

1	0	0	1	1	1	0	1
disp							

Operation: (A), (addr) ← (addr) + 1, addr = 0FF00 + disp

The contents of the memory location specified by addr are decremented by 1 and then the result is loaded into the accumulator and addr.

Label may be specified as a constant, symbol, or expression. Its value must be in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	8
Addressing:	Direct
Flags Affected:	None

Example:

```
DLD     A, TEMP          ;Direct
```

Note: This is an interlocked instruction. If the effective address points to external memory, the processor will not release the bus between the read and write cycles. This allows effective implementation of semaphores.

3.21 DIVIDE

Formats:

```
DIV     EA,T             ;Implied
```

0	0	0	0	1	1	0	1

Operation: Quotient: (EA) ← [(EA / (T)] 15:0

The contents of the concatenated extension register and accumulator are divided by the contents of the temporary register. Division is performed using two signed 16-bit operands, both of which must be positive, and yields a signed 16-bit quotient in the concatenated extension register and accumulator and 16-bit partial remainder. The partial remainder which is returned in the T register is not a true remainder, and for most purposes should be treated as "undefined".

The DIV instruction may be used with either signed or unsigned numbers, so long as the proper range of operand values is observed. In both cases, the divisor must be in the range of 1 to 32767. When using unsigned numbers, the dividend may be any possible value (0 to 65535) and the quotient will be in the same range. Using signed numbers, the dividend must be in the range 0 to 32767, and the quotient will be in the same range.

Cycles:	42
Addressing:	Implied
Flags Affected:	CY/L, OV

Example:

```
DIV  EA,T            ;Implied
```

3.22 EXCLUSIVE-OR EXTENSION WITH ACCUMULATOR

Formats:

X OR A,E ;Implied

0	1	1	0	0	0	0	0

Operation: (A) ← (A)⊻(E)

The contents of the extension register are XORed with the contents of the accumulator. The result is stored in the accumulator.

The truth table for this instruction is shown below.

A b	E b	A b⊻E b
0	0	0
0	1	1
1	0	1
1	1	0

b = bits 7 through 0

Cycles: 4

Addressing: Implied
Flags Affected: None

Example:

XOR A,E ;Implied

3.23 EXCLUSIVE-OR IMMEDIATE WITH ACCUMULATOR

Formats:

X OR A, =data ;Immediate

1	1	1	0	0	1	0	0
data							

Operation: (A) ← (A)⊻data

The data in byte 2 of the instruction are XORed with the contents of the accumulator. The result is stored in the accumulator.

Data must be specified as a literal.

The truth table for this instruction is shown below.

A b	data b	A b ⊻ data b
0	0	0 (*Ab*)
0	1	1 (Ab)
1	0	1 (*Ab*)
1	1	0 (Ab)

b = bits 7 through 0

Cycles: 5
Addressing: Immediate
Flags Affected: None

Example:

```
XOR     A,0DF                   ;Immediate
```

Note: This instruction can be used to selectively complement bits of A

3.24 EXCLUSIVE-OR MEMORY WITH ACCUMULATOR

Formats:

```
XOR     A,label                 ;PC relative, PC is assumed
XOR     A,disp,r                ;PC or PTR relative, r = PC, SP, P2, or P3
XOR     A,@disp,r               ;Auto-indexed, r = P2 or P3
XOR     A,disp,@r               ;Auto-indexed, r = P2 or P3
```

1	1	1	0	0	@	r
disp						

Operation: (A) ← (A) ⊻ (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" are XORed with the contents of the accumulator. The result is stored in the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

The truth table for this instruction is shown below.

A b	addr b	A b ⊻ addr b
0	0	0
0	1	1
1	0	1
1	1	0

b = bits 7 through 0

Cycles:	7, 9 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
XOR     A,MASK        ;PC relative
XOR     A,0C,PC       ;PC relative
XOR     A,-35,P3      ;P3 relative
XOR     A,@-2,P2      ;Auto-indexed by P2
XOR     A,-2,@P3      ;Auto-indexed by P3
```

3.25 EXCLUSIVE-OR MEMORY DIRECT WITH ACCUMULATOR

Formats:

```
XOR     A,label       ; Direct 0FF00 ≤ label ≤ 0FFFF
```

1	1	1	0	0	1	0	1
disp							

Operation: (A) ← (A) ⊻ (addr); addr = 0FF00 + disp

The contents of memory location "addr" are XORed with the contents of the accumulator. The result is stored in the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

The truth table for this instruction is shown below.

A b	addr b	A b ⊻ addr b
0	0	0
0	1	1
1	0	1
1	1	0

b = bits 7 through 0

Cycles:	7
Addressing:	Direct
Flags Affected:	None

Example:

```
XOR     A,BUFFER      ;Direct
```

3.26 EXCHANGE ACCUMULATOR AND EXTENSION REGISTER

Formats:

XCH	E,A	;Implied
XCH	A,E	;Implied

0	0	0	0	0	0	0	1

Operation: (A) ↔ (E)

The contents of the accumulator are exchanged with the contents of the extension register.

Cycles:	5
Addressing:	Implied
Flags Affected:	None

Example:

XCH	E,A	;Implied
XCH	A,E	;Implied

3.27 EXCHANGE EXTENDED ACCUMULATOR AND POINTER REGISTER

Formats:

XCH	r,EA	;Implied, r = PC, SP, P2, or P3
XCH	EA,r	;Implied, r = PC, SP, P2, or P3

0	1	0	0	1	1	r

Operation: (r) ↔ (EA)

The contents of register r are exchanged with the contents of the concatenated extension register and accumulator.

Register r is specified as PC, SP, P2, or P3.

Cycles:	7
Addressing:	Implied
Flags Affected:	None

Examples:

XCH	P2,EA	;Implied
XCH	EA,SP	;Implied

3.28 INCREMENT AND LOAD

Formats:

	A,label	;PC relative, PC is assumed
ILD	A,disp,r	;PC or PTR relative, r = PC, SP, P2, or P3
ILD	A,@disp,r	;Auto-indexed, r = P2 or P3
	A,disp,@r	;Auto-indexed, r = P2 or P3

1	0	0	1	0	@	r
disp						

Operation: (A), (addr) ← (addr) + 1, addr = (r) + disp
If auto-index; (r) ← (r) + disp

The contents of the memory location specified by addr are incremented by 1 and then the result is loaded into the accumulator and addr. Auto-indexing, if specified, is then performed.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	8, 10 if auto-indexed
Addressing:	PC relative, PTR relative, auto-indexed
Flags Affected:	None

Examples:

ILD	A,0F,PC	;PC relative
ILD	A,2,P2	;P2 relative
ILD	A,@1,P3	;Auto-index by P3

3.29 INCREMENT AND LOAD DIRECT

Formats:

ILD	A,label	;Implied 0FF00 ≤ label ≤ 0FFFF

1	0	0	1	0	1	0	1
disp							

Operation: (A), (addr) ← (addr) + 1; addr = 0FF00 + disp

The contents of the memory location specified by addr are incremented by 1 and then the result is loaded into the accumulator and addr.

Label may be specified as a constant, symbol, or expression. Its value must be in the range, 0FF00 through 0FFC0.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	8
Addressing:	Direct
Flags Affected:	None

Example:

```
ILD     A,COUNT     ; Direct
```

3.30 JUMP TO SUBROUTINE

Formats:

```
JSR     label                   ;Absolute
```

0	0	1	0	0	0	0	0
low-order address byte							
high-order address byte							

Operation: ((SP) - 1) ← (PCH) ← byte 3;
((SP) - 2) ← (PCL) ← byte 2;
(SP) ← (SP) - 2

The contents of the high-order 8 bits of the program counter are loaded into the memory location whose address is specified by the contents of the stack pointer −1. Then the immediate data in byte 3 of the instruction are loaded into the high-order bits of the program counter. The contents of the low-order 8 bits of the program counter are loaded into the memory location whose address is specified by the contents of the stack pointer −2. Then the immediate data in byte 2 of the instruction are loaded into the low-order 8 bits of the program counter. The contents of the stack pointer are decremented by 2.

The address assembled into the instruction is label - 1 so that the subroutine at address "label" will be executed.

Cycles:	16
Addressing:	Absolute
Flags Affected:	None

Examples:

```
JSR     PUTL        ;Absolute
JSR     SUBR 1      ;Absolute
```

3.31 JUMP UNCONDITIONAL

Formats:

```
JMP     label                   ;Absolute
```

0	0	1	0	0	1	0	0
Low-order address byte							
high-order address byte							

Operation: (PCH) ← (byte 3); (PCL) ← (byte 2)

Program execution continues at the memory location following that specified in byte 3 and byte 2 of the instruction.

The address assembled into the instruction is label - 1 so that the instruction at address "label" will be the next instruction executed.

Cycles:	9
Addressing:	Absolute
Flags Affected:	None

Examples:

```
JMP     INIT                    ;Absolute
JMP     START+2                 ;Absolute
```

3.32 LOAD ACCUMULATOR TO EXTENSION REGISTER

Formats:

```
LD      E,A                     ;Implied
```

0	1	0	0	1	0	0	0

Operation: (E) ← (A)

The contents of the accumulator replace the contents of the extension register.

Cycles:	4
Addressing:	Implied
Flags Affected:	None

Example:

```
LD      E,A                     ;Implied
```

3.33 LOAD ACCUMULATOR TO STATUS REGISTER

Formats:

LD S,A ;Implied

0	0	0	0	0	1	1	1

Operation: (S) ← (A)

The contents of the accumulator replace the contents of the status register.

Cycles: 3
Addressing: Implied
Flags Affected: None

Example:

LD S,A ;Implied

3.34 LOAD EXTENSION TO ACCUMULATOR

Formats:

LD A,E ;Implied

0	1	0	0	0	0	0	0

Operation: (A) ← (E)

The contents of the extension register replace the contents of the accumulator.

Cycles: 4
Addressing: Implied
Flags Affected: None

Example:

LD A,E ;Implied

3.35 LOAD EXTENDED ACCUMULATOR TO POINTER REGISTER

Formats:

LD r,EA ;Implied,
r = PC, SP, P2, or P3

0	1	0	0	0	1		r

Operation: (r) ← (EA)

The contents of the concatenated extension register and accumulator replace the contents of register r.

Register r is specified as PC, SP, P2, or P3.

Cycles: 5
Addressing: Implied
Flags Affected: None

Examples:

```
LD      P3,EA        ;Implied
LD      SP,EA        ;Implied
```

3.36 LOAD ACCUMULATOR TO TEMPORARY REGISTER

Formats:

```
LD      T,EA         ;Implied
```

0	0	0	0	1	0	0	1

Operation: (T) ← (EA)

The contents of the concatenated extension register and accumulator replace the contents of the temporary register.

Cycles: 4
Addressing: Implied
Flags Affected: None

Example:

```
LD      T,EA         ;Implied
```

3.37 LOAD IMMEDIATE TO ACCUMULATOR

Formats:

```
LD      A,=data      ;Immediate
```

1	1	0	0	0	1	0	0
data							

Operation: (A) ← data

The data in byte 2 of the instruction replace the contents of the accumulator.

Data must be specified as a literal.

Cycles: 5
Addressing: Immediate
Flags Affected: None

Example:

```
LD      A,=0E4       ;Immediate
```

3.38 LOAD IMMEDIATE TO EXTENDED ACCUMULATOR

Formats:

```
LD      EA, = data          ;Immediate
```

1	0	0	0	0	1	0	0
data (low byte)							
data (high byte)							

Operation: (EA) ← data + 1, data

The data in byte 3 and byte 2 of the instruction replace the contents of the concatenated extension register and accumulator.

Data must be specified as a literal.

Cycles:	8
Addressing:	Double precision immediate
Flags Affected:	None

Example:

```
LD      EA, = 0F0F0         ;Immediate
LD      EP, = BUFFER        ;Load EA with the address of
                            BUFFER
```

3.39 LOAD IMMEDIATE TO POINTER REGISTER

Formats:

```
LD      P, = data           ; Implied, r = SP, P2 or P3
```

0	0	1	0	0	1	r
data (low byte)						
data (high byte)						

Operation: (r) ← data + 1, data

The data in byte 3 and byte 2 of the instruction replace the contents of the selected pointer register.

Data must be specified as a literal.

Cycles:	9
Addressing:	Double precision immediate
Flags Affected:	None

Examples:

```
LD      r, = 0F0F0          ; Immediate
LD      r, = DBUF+5         ; Load r with the address of DBUF+5
```

3.40 LOAD IMMEDIATE TO TEMPORARY REGISTER

Formats:

```
LD      T, = data          ;Double precision immediate
```

1	0	1	0	0	1	0	0
data (low byte)							
data (high byte)							

Operation: (T) ← data + 1, data

The data in byte 3 and byte 2 of the instruction replace the contents of the temporary register.

Data must be specified as a literal.

Cycles:	8
Addressing:	Double precision immediate
Flags Affected:	None

Examples:

```
LD      T, = 0F0F0         ;Immediate
LD      T, = DBUF + 5      ;Load T with the address of
                           DBUF + 5
```

3.41 LOAD MEMORY TO ACCUMULATOR

Formats:

```
LD      A,label            ;PC relative, PC is assumed
LD      A,disp,r           ;PC or PTR relative,
                           r = PC, SP, P2, or P3
LD      A,@disp,r          ;Auto-indexed, r = P2 or P3
LD      A,disp,@r          ;Auto-indexed, r = P2 or P3
```

1	1	0	0	0	@	r
disp						

Operation: (A) ← (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" replace the contents of the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	7, 9 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
LD      A,0FF,PC          ;PC relative
LD      A,START+3         ;PC relative
LD      A,BUFF+2,P2       ;P2 relative
LD      A,@TWO,P2         ;Auto-indexed by P2
LD      A,10,@P3          ;Auto-indexed by P3
```

3.42 LOAD MEMORY TO EXTENDED ACCUMULATOR

Formats:

```
LD      EA,label          ;PC relative, PC is assumed
LD      EA,disp,r         ;PC or PTR relative,
                          r = PC, SP, P2, or P3
LD      EA,@disp,r        ;Auto-indexed, r = P2 or P3
LD      EA,disp,@r        ;Auto-indexed, r = P2 or P3
```

1	0	0	0	0	@	r
disp						

Operation: (EA) ← (addr + 1, addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory locations "addr + 1" and "addr" replace the contents of the concatenated extension register and accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	10, 12 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
LD      EA,TEMP           ;PC relative
LD      EA,0F,PC          ;PC relative
LD      EA,INCR,SP        ;SP relative
LD      EA,@VAL,P3        ;Auto-indexed by P3
LD      EA,1,@P2          ;Auto-indexed by P2
```

3.43 LOAD MEMORY TO TEMPORARY REGISTER

Formats:

```
LD      T,label          ;PC relative, PC is assumed
LD      T,disp,r         ;PC or PTR relative,
                         r = PC, SP, P2, or P3
LD      T,@disp,r        ;Auto-indexed, r = P2 or P3
LD      T,disp,@r        ;Auto-indexed, r = P2 or P3
```

1	0	1	0	0	@	r
disp						

Operation: (T) ← (addr + 1, addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory locations "addr + 1" and "addr" replace the contents of the temporary register.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	10, 12 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
LD      T,BUF+10         ;PC relative
LD      T,0FF,PC         ;PC relative
LD      T,INCR,SP        ;SP relative
LD      T,@VAL,P3        ;Auto-indexed by P3
LD      T,X'F,@P2        ;Auto-indexed by P2
```

3.44 LOAD MEMORY DIRECT TO ACCUMULATOR

Formats:

```
LD      A,label          ; Direct 0FF00 ≤ label ≤ 0FFFF
```

1	1	0	0	0	1	0	1
disp							

Operation: (A) ← (addr); addr = 0FF00 + disp

The contents of memory location "addr" replace the contents of the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must be in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	7
Addressing:	Direct
Flags Affected:	None

Example:

```
LD      A,BUFFER+4          ;Direct
```

3.45 LOAD MEMORY DIRECT TO EXTENDED ACCUMULATOR

Formats:

```
LD      EA,label            ; Direct 0FF00 ≤ label ≤ 0FFFF
```

1	0	0	0	0	1	0	1
disp							

Operation: (EA) ← (addr + 1, addr); addr = 0FF00 + disp

The concatenated extension register and accumulator is loaded from memory locations "addr + 1" and "addr."

Label may be specified as a constant, symbol, or expression. Its value must be in the range,0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	10
Addressing:	Double precision direct
Flags Affected:	None

Example:

```
LD      EA,TEMP             ;Direct
```

3.46 LOAD MEMORY DIRECT TO TEMPORARY REGISTER

Formats:

LD	T,label	;Direct 0FF00 ≤ label ≤ 0FFFF

1	0	1	0	0	1	0	1
disp							

Operation: (T) ← (addr + 1, addr); addr = 0FF00 + disp

The temporary register is loaded from memory locations "addr + 1 and addr."

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	10
Addressing:	Double precision direct
Flags Affected:	None

Example:

LD	T,TEMP	;Direct

3.47 LOAD POINTER REGISTER TO EXTENDED ACCUMULATOR

Formats:

LD	EA,r	;Implied, r = PC, SP, P2, or P3

0	0	1	1	0	0	r	

Operation: (EA) ← (r)

The contents of register r replace the contents of the concatenated extension register and accumulator.

Register r is specified as PC, SP, P2, or P3.

Cycles:	4
Addressing:	Implied
Flags Affected:	None

Examples:

LD	EA,P3	;Implied
LD	EA,PC	;Implied

3.48 LOAD STATUS REGISTER TO ACCUMULATOR

Formats:

```
LD      A,S                     ;Implied
```

0	0	0	0	0	1	1	0

Operation: (A) ← (S)

The contents of the status register replace the contents of the accumulator.

Cycles:	3
Addressing:	Implied
Flags Affected:	None

Example:

```
LD      A,S                     ;Implied
```

3.49 LOAD TEMPORARY REGISTER TO EXTENDED ACCUMULATOR

Formats:

```
LD      EA,T                    ;Implied
```

0	0	0	0	1	0	1	1

Operation: (EA) ← (T)

The contents of the temporary register replace the contents of the concatenated extension register and accumulator.

Cycles:	4
Addressing:	Implied
Flags Affected:	None

Example:

```
LD      EA,T                    ;Implied
```

3.50 MULTIPLY

Formats:

MPY EA,T ;Implied

0	0	1	0	1	1	0	0

Operation: (EA) ← [EA] * (T) 31:16,
(T) ← [(EA) * (T)] 15:0

The contents of the concatenated extension register and accumulator are multiplied by the contents of the temporary register. Multiplication is performed using two signed 16-bit operands (the multiplier must be positive) and yields a signed 32-bit result.

Cycles:	37
Addressing:	Implied
Flags Affected:	CY, OV

Example:

MPY EA,T ;Implied

3.51 NO OPERATION

Formats:

NOP ;Implied

0	0	0	0	0	0	0	0

Operation: (PC) ← (PC) + 1

The contents of the program counter are incremented by 1 and execution continues at the next instruction. No other operation is performed.

This instruction is useful for adding delays to timing loops to make up exact time delays. It is also useful when writing a program to add a few NOP's throughout the code so that instructions can be added without having to reassemble your program. Then on the last assembly pass, the extra NOP's can be removed.

Cycles:	3
Addressing:	Implied
Flags Affected:	None

Example:

NOP ;Implied

3.52 OR EXTENSION WITH ACCUMULATOR

Formats:

OR A,E ;Implied

0	1	0	1	1	0	0	0

Operation: (A) ← (A) ∨ (E)

The contents of the extension register are ORed with the contents of the accumulator. The result is stored in the accumulator.

The truth table for this instruction is shown below.

A b	E b	A b ∨ E b
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7 through 0

Cycles: 4
Addressing: Implied
Flags Affected: None

Example:

OR A,E ;Implied

3.53 OR IMMEDIATE WITH ACCUMULATOR

Formats:

OR A, =data ;Immediate

1	1	0	1	1	1	0	0
data							

Operation: (A) ← (A) ∨ data

The data in byte 2 of the instruction are ORed with the contents of the accumulator. The result is stored in the accumulator.

Data must be specified as a literal.

The truth table for this instruction is shown below.

A b	data b	A b ∨ data b
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7 through 0

Cycles: 7
Addressing: Immediate
Flags Affected: None

Example:

```
OR      A,0F            ;Immediate
```

3.54 OR IMMEDIATE WITH STATUS REGISTER

Formats:

```
OR      S, = data       ;Immediate
```

0	0	1	1	1	0	1	1
data							

Operation: (S) ← (S) ∨ data

The data in byte 2 of the instruction are ORed with the contents of the status register. The result is stored in the status register.

Data must be specified as a literal.

The truth table for this instruction is shown below.

S b	data b	S b ∨ data b
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7, 6, 3, 2, 1, 0

Cycles: 5
Addressing: Immediate
Flags Affected: None

Example:

```
OR      S, = DATA       ;Immediate
```

3.55 OR MEMORY WITH ACCUMULATOR

Formats:

```
OR      A,label          ;PC relative, PC is assumed
OR      A,disp,r         ;PC or PTR relative,
                         r = PC, SP, P2, or P3
OR      A,@disp,r        ;Auto-indexed, r = P2 or P3
OR      A,disp,@r        ;Auto-indexed, r = P2 or P3
```

1	1	0	1	1	@	r
disp						

Operation: (A) ← (A)∨ (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" are ORed with the contents of the accumulator. The result is stored in the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

The truth table for this instruction is shown below.

A b	addr b	A b∨ addr b
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7 through 0

Cycles: 7, 9 if auto-indexed
Addressing: PC relative, pointer relative, auto-indexed
Flags Affected: None

Examples:

```
OR      A,DATE+2         ;PC relative
OR      A,02F,PC         ;PC relative
OR      A,TEMP,P2        ;P2 relative
OR      A,@10,P2         ;Auto-indexed by P2
OR      A,19,@P2         ;Auto-indexed by P2
```

3.56 OR MEMORY DIRECT WITH ACCUMULATOR

Formats:

```
OR      A,label                 ;Direct 0FF00 ≤ label
                                ≤ 0FFFF
```

1	1	0	1	1	1	0	1
disp							

Operation: (A) ← (A)∨ (addr); addr = 0FF00 + disp

The contents of memory location "addr" are ORed with the contents of the accumulator. The result is stored in the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM. The truth table for this instruction is shown below.

A b	addr b	A b ∨ addr b
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7 through 0

Cycles: 7
Addressing: Direct
Flags Affected: None

Example:

```
OR      A,TEMP                  ;Direct
```

3.57 POP STACK TO ACCUMULATOR

Formats:

```
POP     A                       ;Implied
```

0	0	1	1	1	0	0	0

Operation: (A) ← ((SP)); (SP) ← (SP) + 1

The contents of the memory location specified by the top of the stack are loaded into the accumulator. The contents of the stack pointer are incremented by 1.

Cycles: 6
Addressing: Implied
Flags Affected: None

Example:

```
POP     A                       ;Implied
```

3.58 POP STACK TO EXTENDED ACCUMULATOR

Formats:

```
POP     EA                      ;Implied
```

0	0	1	1	1	0	1	0

Operation: (A) ← ((SP)); (E) ← ((SP) + 1):
(SP) ← (SP) + 2

The contents of the memory location whose address is specified by the contents of the stack pointer are loaded into the accumulator. The contents of the memory location specified by the contents of the stack pointer + 1 are loaded into the extension register. The contents of the stack pointer are incremented by 2.

Cycles:	9
Addressing:	Implied
Flags Affected:	None

Example:

```
POP     EA                      ;Implied
```

3.59 POP STACK TO POINTER REGISTER

Formats:

```
POP     r                       ;Implied, r = P2 or P3
```

0	1	0	1	1	1	1	r

Operation: (rL) ← ((SP)); (rH) ← ((SP) + 1);
(SP) ← (SP) + 2

The contents of the memory location whose address is specified by the contents of the stack pointer are loaded into the low-order 8 bits of the specified register. The contents of the memory location specified by the contents of the stack pointer + 1 are loaded into the high-order 8 bits of the specified register. The contents of the stack pointer are incremented by 2.

Register r is specified as P2 or P3.

Cycles:	10
Addressing:	Implied

Examples:

```
POP     P2                      ;Implied
POP     P3                      ;Implied
```

3.60 PUSH ACCUMULATOR TO STACK

Formats:

PUSH A ;Implied

0	0	0	0	1	0	1	0

Operation: (SP) ← (SP) − 1; ((SP)) ← (A)

The contents of the stack pointer are decremented by 1 and the contents of the accumulator are loaded into the new top of the stack.

Cycles:	5
Addressing:	Implied
Flags Affected:	None

Example:

PUSH A ;Implied

3.61 PUSH EXTENDED ACCUMULATOR TO STACK

Formats:

PUSH EA ;Implied

0	0	0	0	1	0	0	0

Operation: ((SP) − 1) ← (E); ((SP) − 2) ← (A);
(SP) ← (SP) − 2

The contents of the extension register are loaded into the memory location whose address is specified by the contents of the stack pointer − 1. The contents of the accumulator are loaded into the memory location whose address is specified by the contents of the stack pointer − 2. The contents of the stack pointer are decremented by 2.

Cycles:	8
Addressing:	Implied
Flags Affected:	None

Example:

PUSH EA ;Implied

3.62 PUSH POINTER REGISTER TO STACK

Formats:

PUSH r ;Implied, r = PC, P2, or P3

0	1	0	1	0	1	r

Operation: ((SP) − 1) ← (rH); ((SP) − 2) ← (rL);
(SP) ← (SP) − 2

The contents of the high-order 8 bits of the specified register are loaded into the memory location whose address is specified by the contents of the stack pointer − 1. The contents of the low-order 8 bits of the specified register are loaded into the memory location whose address is specified by the contents of the stack pointer − 2. The contents of the stack pointer are decremented by 2.

Register r is specified as PC, P2, or P3.

Cycles:	8
Addressing:	Implied
Flags Affected:	None

Examples:

PUSH	P3	;Implied
PUSH	PC	;Implied
PUSH	P2	;Implied

3.63 PUSH AND LOAD POINTER REGISTER IMMEDIATE

Formats:

PLI r, = data ;Double precision immediate
r = P2, P3

0	0	1	0	0	0	1	r
data (low byte)							
data (high byte)							

Operation: ((SP) − 1) ← (rH) ← data + 1;
((SP) − 2) ← (rL) ←data;
(SP) ← (SP) − 2

The contents of the high-order 8 bits of the specified register are loaded into the memory location whose address is specified by the contents of the stack pointer − 1. Then the immediate data in byte 3 of the instruction are loaded into the high-order 8 bits of the specified register.The contents of the low-order 8 bits of the specified register are copied to the memory location whose address is specified by the contents of the stack pointer − 2. Then the immediate data in byte 2 of the instruction are loaded into the low-order 8 bits of the specified register. The contents of the stack pointer are decremented by 2.

Register r is specified as P2 or P3.
Data must be specified as a literal.

Cycles:	16
Addressing:	Immediate
Flags Affected:	None

Examples:

```
PLI     P2, = DATA        ;Immediate
PLI     P3, = X'FF00      ;Immediate
```

3.64 RETURN

Formats:

```
RET                       ;Implied
```

0	1	0	1	1	1	0	0

Operation: (PCL) ← ((SP)); (PCH) ← ((SP) + 1);
(SP) ← (SP) + 2

The contents of the memory location whose address is specified by the contents of the stack pointer are loaded into the low-order 8 bits of the program counter. The contents of the memory location specified by the contents of the stack pointer + 1 are loaded into the high-order 8 bits of the program counter.The contents of the stack pointer are incremented by 2.

Cycles:	10
Addressing:	Implied

Example:

```
RET                       ;Implied
```

3.65 ROTATE ACCUMULATOR RIGHT

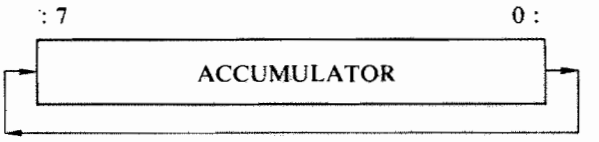
Formats:

```
RR      A                 ;Implied
```

0	0	1	1	1	1	1	0

Operation: (Ai) → (Ai-1); i = 7, 1; A0 → A7

The contents of the accumulator are rotated right one bit position. The initial content of bit 0 is shifted into bit 7.



Cycles:	3
Addressing:	Implied
Flags Affected:	None

Example:

```
RR      A                 ;Implied
```

3.66 ROTATE ACCUMULATOR RIGHT WITH LINK

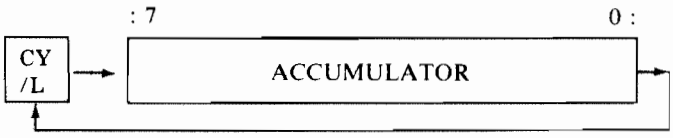
Formats:

RRL A ;Implied

0	0	1	1	1	1	1	1

Operation: (Ai) → (Ai - 1); i = 7, 1; A0 → CY/L → A7

The contents of the accumulator are rotated right one bit position. The initial content of bit 0 is rotated into the carry/link bit. The initial content of the carry/link bit is rotated into accumulator bit 7.



Cycles: 3
Addressing: Implied
Flags Affected: None

Example:

RRL ;Implied

3.67 SEARCH AND SKIP IF CHARACTER MATCH

Formats:

SSM r ;Indexed, r = P2 or P3

0	0	1	0	1	1	1	r

Operation: See below

The SSM instruction searches memory for a character that matches the one in the accumulator. The address where the search begins is specified by P2 or P3. The maximum number of memory locations searched is 256. The search is conducted by using the specified pointer to indicate the address of the character to be tested. The pointer is incremented by 1 after each test. If a match is found within the maximum number of searches allowed, the next two program bytes are skipped. In this case, the instruction is concluded with the specified pointer pointing to an address one greater than the address for which the match occurred.

If a match is not found within the maximum number of searches allowed, the next instruction in sequence will be executed. In this case, the specified pointer will have incremented by one less than the maximum number of searches allowed.

Cycles: Variable: If match: 7 + (4 *N)
If no match: 5 + (4 *N) or 1027 for limit = 256 where N = #of unsuccessful character comparisons

Addressing: Indirect
Flags Affected: None

Examples:

SSM P2 ;Indexed
SSM P3 ;Indexed

3.68 SHIFT ACCUMULATOR LEFT

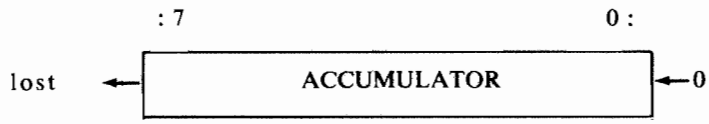
Formats:

SL A ;Implied

0	0	0	0	1	1	1	0

Operation: (Ai) ← (Ai − 1); i = 7, 1; A0 ← 0

The contents of the accumulator are shifted left one bit position. The initial content of bit 7 is shifted out and lost. A zero is shifted into bit 0.



Cycles: 3
Addressing: Implied
Flags Affected: None

Example:

SL A ;Implied

3.69 SHIFT ACCUMULATOR RIGHT

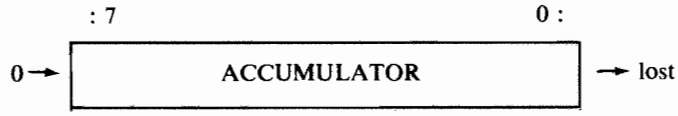
Formats:

SR A ;Implied

0	0	1	1	1	1	0	0

Operation: (Ai) → (Ai − 1); i = 7, 1; 0 → A7

The contents of the accumulator are shifted right one bit position. A zero is shifted into bit 7. The initial content of bit 0 is shifted out and lost.



Cycles: 3
Addressing: Implied
Flags Affected: None

Example:

SR A ;Implied

3.70 SHIFT ACCUMULATOR RIGHT WITH LINK

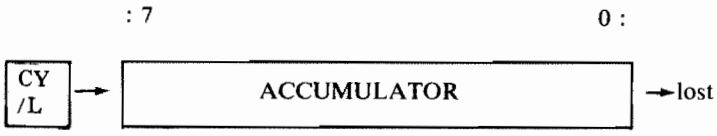
Formats:

SRL A ;Implied

0	0	1	1	1	1	0	1

Operation: (Ai) → (AI - 1); i = 7, 1; CY/L → A7

The contents of the accumulator are shifted right one bit position. The content of the carry/link bit is shifted into accumulator bit 7. The initial content of accumulator bit 0 is shifted out and lost.



Cycles: 3
Addressing: Implied
Flags Affected: None

Example:

SRL A ;Implied

3.71 SHIFT EXTENDED ACCUMULATOR LEFT

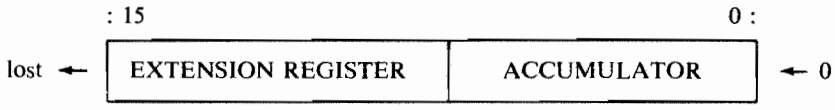
Formats:

SL EA ;Implied

0	0	0	0	1	1	1	1

Operation: (EAi) ← (EAi – 1); i = 15, 1; A0 ← 0

The contents of the concatenated extension register and accumulator are shifted left one bit position. The initial content of bit 15 of the extension register is shifted out and lost. A zero is shifted into bit position 0 of the accumulator.



Cycles: 4
Addressing: Implied
Flags Affected: None

Example:

SL EA ;Implied

3.72 SHIFT EXTENDED ACCUMULATOR RIGHT

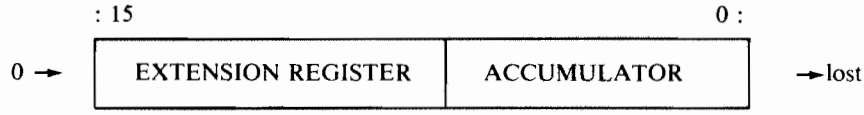
Formats:

SR EA ;Implied

0	0	0	0	1	1	0	0

Operation: (EAi) → (EAi – 1); i = 15, 1; 0 → A15

The contents of the concatenated extension register and the accumulator are shifted right one bit position. A zero is shifted into bit 15 of the extension register. The initial content of bit 0 of the accumulator is shifted out and lost.



Cycles:	4
Addressing:	Implied
Flags Affected:	None

Example:

```
SR      EA                  ;Implied
```

3.73 STORE ACCUMULATOR IN MEMORY

Formats:

```
                A,label             ;PC relative, PC is assumed
                A,disp,r            ;PC or PTR relative, r =
                                    PC, SP, P2, or P3
ST
                A,@disp,r           ;Auto-indexed, r = P2 or P3
                A,disp,@r           ;Auto-indexed, r = P2 or P3
```

1	1	0	0	1	@	r
disp						

Operation: (A) → (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of the accumulator replace the contents of memory location "addr".

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	7, 9 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
ST      A,TEMP              ;PC relative
ST      A,0F,P3             ;P3 relative
ST      A,@ -8,P2           ;Auto-indexed by P2
ST      A, -8,@P3           ;Auto-indexed by P3
```

3.74 STORE ACCUMULATOR IN MEMORY DIRECT

Formats:

ST	A,label	;Direct 0FF00 ≤ label ≤ 0FFFF

1	1	0	0	1	1	0	1
disp							

Operation: (A) → (addr); addr = 0FF00 + disp

The contents of the accumulator replace the contents of the memory location "addr".

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	7
Addressing:	Direct
Flags Affected:	None

Example:

ST	A,BUFFER	;Direct

3.75 STORE EXTENDED ACCUMULATOR IN MEMORY

Formats:

	EA,label	;PC relative, PC is assumed
	EA,disp,r	;PC or PTR relative, r = PC, SP, P2, or P3
ST		
	EA,@disp,r	;Auto-indexed, r = P2 or P3
	EA,disp,@r	;Auto-indexed, r = P2 or P3

1	0	0	0	1	@	r
disp						

Operation: (EA) → (addr + 1, addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of the concatenated extension register and accumulator replace the contents of memory locations "addr + 1" and "addr."

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

Cycles:	10, 12 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	None

Examples:

```
ST      EA,TEMP        ;PC relative
ST      EA,0A02        ;PC relative
ST      EA,DATA,SP     ;SP relative
ST      EA,@10,P3      ;Auto-indexed by P3
ST      EA,-20,@P3     ;Auto-indexed by P3
```

3.76 STORE EXTENDED ACCUMULATOR IN MEMORY DIRECT

Formats:

```
ST      EA,label       ; Direct 0FF00 ≤ label ≤ 0FFFF
```

1	0	0	0	1	1	0	1
disp							

Operation: (EA) → (addr + 1, addr); addr = 0FF00 + disp

The contents of the concatenated extension register and accumulator replace the contents of memory locations addr + 1 and "addr."

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

Cycles:	10
Addressing:	Double precision direct
Flags Affected:	None

Example:

```
ST      EA,0FF2D       ;Direct
```

3.77 SUBTRACT EXTENSION FROM ACCUMULATOR

```
SUB     A,E                     ;Implied
```

0	1	1	1	1	0	0	0

Operation: (A) ← (A) — (E)

The contents of the extension register are subtracted from the contents of the accumulator. The result is stored in the accumulator.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of the extension register and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	4
Addressing:	Implied
Flags Affected:	CY/L, OV

Example:

```
SUB     A,E                     ;Implied
```

3.78 SUBTRACT IMMEDIATE FROM ACCUMULATOR

Formats:

```
SUB     A, =data                ;Immediate
```

1	1	1	1	1	1	0	0
data							

Operation: (A) ← (A) − data

The data in byte 2 of the instruction are subtracted from the contents of the accumulator. The result is stored in the accumulator.

Data must be specified as a literal.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of data and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	5
Addressing:	Immediate
Flags Affected:	CY/L, OV

Example:

```
SUB     A, =BUFFER              ;Immediate
```

3.79 SUBTRACT IMMEDIATE FROM EXTENDED ACCUMULATOR

Formats:

```
SUB     EA,=data            ;Immediate
```

1	0	1	1	1	1	0	0
data (low byte)							
data (high byte)							

Operation: (EA) ← (EA) − data + 1, data

The data in byte 3 and byte 2 of the instruction are subtracted from the contents of the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Data must be specified as a literal.

CY/L is set if a carry from bit 7 of the extension register occurred; otherwise CY/L is clared. OV is set if the sign of the result is the same as the sign of the immediate data and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	8
Addressing:	Immediate, double precision
Flags Affected:	CY/L, OV

Example:

```
SUB     EA,=0FFFF           ;Immediate
```

3.80 SUBTRACT MEMORY FROM ACCUMULATOR

Formats:

```
SUB     A,label             ;PC relative, PC is assumed
SUB     A,disp,r            ;PC or PTR relative, r = PC, SP, P2, or P3

SUB     A,@disp,r           ;Auto-indexed, r = P2 or P3
SUB     A,disp,@r           ;Auto-indexed, r = P2 or P3
```

1	1	1	1	1	@	r
disp						

Operation: (A) ← (A) − (addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory location "addr" are subtracted from the contents of the accumulator. The result is stored in the accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of (addr) and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	7, 9 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	CY/L, OV

Examples:

```
SUB     A,DATA+2        ;PC relative
SUB     A,2,PC          ;PC relative
SUB     A,TEMP,P3       ;P3 relative
SUB     A,@05,P2        ;Auto-indexed by P2
SUB     A,-6,@P3        ;Auto-indexed by P3
```

3.81 SUBTRACT MEMORY FROM EXTENDED ACCUMULATOR

Formats:

```
SUB     EA,label        ;PC relative, PC is assumed
SUB     EA,disp,r       ;PC or PTR relative, r = PC, SP, P2, or P3

SUB     EA,@disp,r      ;Auto-indexed, r = P2 or P3
SUB     EA,disp,@r      ;Auto-indexed, r = P2 or P3
```

1	0	1	1	1	@	r
disp						

Operation: (EA) ← (EA) − (addr + 1, addr); addr = (r) + disp
If auto-index: (r) ← (r) + disp

The contents of memory locations "addr + 1" and "addr" are subtracted from the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Disp or label may be specified as a constant, symbol, or expression.

Register r is specified as PC, SP, P2, or P3. If a register is not specified, the assembler first attempts direct addressing and then defaults to PC. Register P2 or P3 must be specified for auto-indexed addressing.

CY/L is set if a carry from bit 7 of the extension reigster occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of (addr) and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	10, 12 if auto-indexed
Addressing:	PC relative, pointer relative, auto-indexed
Flags Affected:	CY/L, OV

Examples:

```
SUB     EA,0ED,PC       ;PC relative
SUB     EA,TEMP         ;PC relative
SUB     EA,2,P2         ;P2 relative
SUB     EA,@-6,P3       ;Auto-indexed by P3
SUB     EA,6,@P2        ;Auto-indexed by P2
```

3.82 SUBTRACT MEMORY DIRECT FROM ACCUMULATOR

Formats:

SUB A,label ;Direct

1	1	1	1	1	1	0	1
disp							

Operation: (A) ← (A) - (addr); addr = 0FF00 + disp

The contents of memory location "addr" are subtracted from the contents of the accumulator. The result is stored in the accumulator.

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

CY/L is set if a carry from bit 7 of the accumulator occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of (addr) and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	7
Addressing:	Direct
Flags Affected:	CY/L, OV

Example:

SUB A, DATA ;Direct

3.83 SUBTRACT MEMORY DIRECT FROM EXTENDED ACCUMULATOR

Formats:

SUB EA, label ;Direct 0FF00 ≤ label

1	0	1	1	1	1	0	1
disp							

Operation: (EA) ← (EA) - (addr + 1, addr); addr = 0FF00 + disp

The contents of memory locations "addr + 1 and addr" are subtracted from the contents of the concatenated extension register and accumulator. The result is stored in the extension register and accumulator.

Label may be specified as a constant, symbol, or expression. Its value must lie in the range, 0FF00 through 0FFFF.

Disp is treated as an unsigned displacement. Since on-chip RAM is located from 0FFC0 through 0FFFF, displacement values of 0C0 through 0FF will address all of the on-chip RAM.

CY/L is set if a carry from bit 7 of the extension register occurred; otherwise CY/L is cleared. OV is set if the sign of the result is the same as the sign of (addr) and opposite the sign of the accumulator; otherwise OV is cleared.

Cycles:	10
Addressing:	Direct, double precision
Flags Affected:	CY/L, OV

Example:

SUB EA, TEMP ;Direct

Chapter 4

ASSEMBLY LANGUAGE

The 70-series assembly language statements have well-defined formats constructed from the elements described in this chapter.

4.1 BASIC ELEMENTS

4.1.1 Character Set

Statements are written using the following letters, numbers, and special characters:

Letters:	A through Z
Numbers:	0 through 9
Special Characters:	! $ % & ' ( ) * + , - . / : ; < = > @ b # ^
	Note: b means blank

Except for the lower-case letters, any of the printable characters from the ASCII character set may be specified with a .ASCII directive (see 5.2.1).

```
Example:    MSG3: .ASCII 'ORDER # 326-001'
```

Nonprintable (or printable) characters may be specified with a .BYTE directive (see 5.2.3) or a .DBYTE directive (see 5.2.4).

```
Examples:   CR:   .BYTE 0D        ;CARRIAGE RETURN
            CRLF: .DBYTE 0D0A     ;CARRIAGE RETURN/LINE FEED
```

4.1.2 Data Representation

Data are represented internally in 807X in twos-complement integer notation and binary-coded-decimal (BCD) notation. In twos-complement notation, the negative of a number is formed by complementing each bit in the data word and adding one to the complemented number. The sign of the number is indicated by the most significant bit. When the most significant bit is a '0', the number is positive or zero; when the most significant bit a '1', the number is negative. Maximum range for a 16-bit number in this system is $7FFF_{16}$ ($+32767_{10}$) to 8000_{16} (-32768_{10}). Maximum range for an 8-bit number is $7F_{16}$ ($+127_{10}$) to 80_{16} (-128_{10}).

BCD notation is a system of number representation in which the decimal digits 0 through 9 are represented by a group of four binary digits. For example:

	7	0
25_{10} =	0 0 1 0	0 1 0 1

The maximum range of BCD values if 0 to 99_{10}. To obtain a BCD constant, the user may code one or two digits (0 through 9) as a hexadecimal constant, for example, 059.

4.1.3 Terms

The relationship of terms is shown in figure 4-1. The various types of terms are described in the following paragraphs.

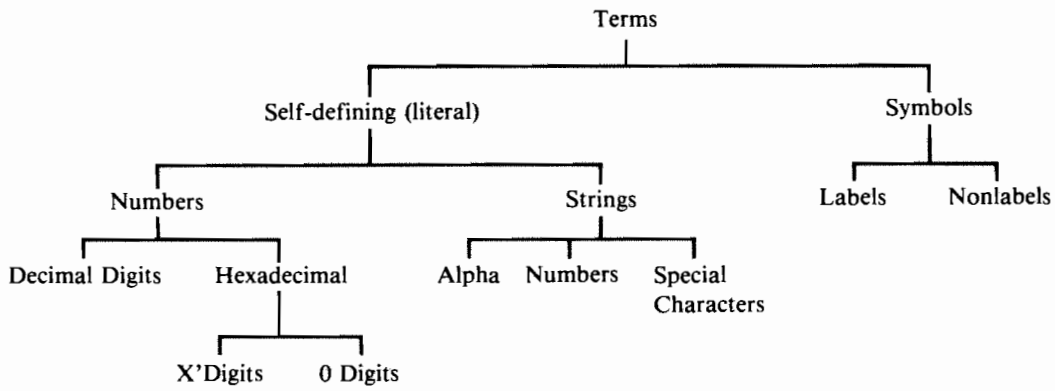


Figure 4 – 1. Relationship of Terms

4.1.3.1 Self-Defining Terms

A self-defining term, or constant, has its value inherent in the term. The assembler program does not assign a value to the term but does derive the value from the term.

Self-defining terms are used to specify immediate data, addresses, registers, and input/output information to the assembler program. Three types of self-defining terms are available: decimal, hexadecimal, and character (or string).

A decimal self-defining term is zero, or a decimal integer that does not begin with zero. For 16-bit data, the value range is 0 to 65,535 for an unsigned decimal integer and −32,768 to +32,767 for a signed decimal integer. For 8-bit data, the value range is 0 to 255 for an unsigned decimal integer and −128 to +127 for a signed decimal integer. It should be noted that having signed and unsigned data is just a coding convenience made available because some instructions treat data as signed values and others treat data as unsigned values. For example, in 8-bit data, a signed number such as -1_{10} or an unsigned number such as 255_{10} is converted to FF_{16} by the assembler. The microprocessor performs signed arithmetic except when executing decimal-add instructions.

Examples:

	decimal constants
.BYTE	200, −100, 0, +32
.DBYTE	0, 40000, −3165
LD	A, =0
XOR	A, =20

A hexadecimal self-defining term may be specified in either of two ways. The term may start with X', or the term may start with a leading zero. The range of hexadecimal numbers is 0 to $FFFF_{16}$ for 16-bit data and 0 to FF_{16} for 8-bit data.

Examples:

	hexadecimal constants
.BYTE	X'FF, X'10
.DBYTE	X'1FE, X'FFFF
XOR	A, =015
LD	A, =X'40

A character self-defining term is defined as a string. A string is a series of characters or a single character enclosed in single quote marks (for example, 'THIS IS A STRING'). All letters, numbers, and special characters (including blanks) may be specified in a string. If a single quote mark is part of the character string, it should immediately be preceded by another single quote mark; for example, 'DON''T DO IT' represents DON'T DO IT. A null string ('') causes the assembler to generate a single blank. String characters are translated to ASCII code in memory with each character occupying 8 bits. Refer to the .ASCII directive described in 5.2.1.

Examples:

	ASCII constants
.ASCII	'NUMBER = '
XOR	XOR A, = 'Y'
LD	LD A, = '?'

4.1.3.2 Symbolic Terms

Symbols are the most common means of referencing address locations or arbitrary values. Symbols are defined (assigned values) by one of two methods:

1. By appearing in a label field in a statement (see 4.2.1).

symbol

```
SUB1:     LD     A, =0                    ;CLEAR AC
```

The value assigned to a symbol appearing in the label field is the address of the instruction, data, or storage location named by the symbol.

2. By using an assignment statement to assign a specific value to a symbol (see 5.1).

symbol

```
P2 = 2                                    ;STACK POINTER
```

Symbol construction must meet the following restrictions:

1. A symbol may contain one or more alphanumeric characters, the first of which must be either a letter or a dollar sign ($).

2. Although up to 32 characters may be included, only the first six characters are recognized by the assembler program. Therefore, the programmer must ensure that a long symbol is unique in the first six characters.

Example: LONGSY, LONGSYMBOL1, LONGSYMBOL2 — are identical to the assembler

3. If the first character in the symbol is a dollar sign ($), the symbol is defined as a local symbol. The .LOCAL operator allows the programmer to specify that local symbols appearing between two .LOCAL directive statements have a certain meaning only within that region of the program (see 5.2.13, .LOCAL Directive). This enables the programmer to use common mnemonics throughout a program without causing a conflict of names.

NOTE: Within a local region, a long local symbol must be unique in the first <u>five</u> characters, including the dollar sign ($).

Example: $ABCD, $ABCDEF — are identical symbols to the assembler.

4. No special characters or embedded blanks may appear within a symbol.

5. Symbol values cannot exceed 65,535 for unsigned 16-bit data, −32, 768 to +32,767 for signed 16-bit data, 255 for unsigned 8-bit data or −128 to +127 for signed 8-bit data.

Several examples of symbols follow:

Legal Symbols	Illegal Symbols	Reason Illegal
$ABC	LONGSYMBOL1	First six characters are not unique
LONGSYMBOL	LONGSYMBOL2	
$AB2	2AB	First character must be a letter or a dollar sign
$2	#CDE	
XYZ	XYZ$	Last character is not alphanumeric
$ABCDEF	$ABCDE	First five characters of the local symbols are not unique.
$ABC2EF	$ABCDF	

A symbol term may represent a memory address 8 bit or 16 bit data and, hence, may have a value ranging between 0 and $65,535_{10}$.

4.1.4 Expressions

Operand entries (see 4.2.3), consisting of either a single term or an arithmetic or a logical combination of terms, are called expressions. Expressions are either simple or multiterm. Simple expressions are single terms, such as a symbol or a self-defining term. Multiterm expressions are simple expressions combined using the arithmetic and logical operators shown in table 4-1. The multiterm expression is evaluated by the assembler program in a left-to-right order regardless of the operators used between the terms. Parentheses are not permitted for the purpose of grouping arithmetic and/or logical operations; they have special significance in defining certain assembler functions.

Examples:

```
TABLE + X'10
100 - 1
ENTRY1 + ENTRY2 - 4
A > B * DISKAD
```

The result of the evaluated expression is a 16-bit value; however, the magnitude of the expression must be compatible with the storage allocated for the expression. For example, if the expression is to be stored in 8 bits, then the value of the expression must not exceed FF_{16}.

Example:

```
LD    A,=(X'40 + CHAR)
```

Expression value must not exceed FF_{16}.

If the expression is used to indicate a register, then the value must not exceed 3.

Example:

```
AND   A0, (PTR + N)
```

Expression value must not exceed 3.

Table 4-1 lists the arithmetic and logical operators available for forming expressions.

A unary operator operates upon one operand and appears in the format 'operator operand' (for example, −9). A binary operator operates upon two operands and appears in the format 'operand1 operator operand2' (for example, A&B).

The expressions "A<B", "A=B", and "A>B" cause the specified comparison to be made. The result is one if the condition is true or zero if the condition is false.

Table 4-1. Arithmetic and Logical Operators

Operator	Function	Type
+	Addition	Binary
–	Subtraction	Unary or binary
*	Multiplication	Binary
/	Division	Binary
%	Logical NOT	Unary
&	Logical AND	Binary
!	Logical OR	Binary
<	"Less Than"	Binary
=	"Equal To"	Binary
>	"Greater Than"	Binary

4.1.5 Reserved Words

Reserved words are these symbols that already have special meaning to the assembler and therefore cannot appear as user-defined symbols.

The following symbols are reserved:

Symbol	Meaning
A	Accumulator
E	Extension register
EA	Extended Accumulator (concatenated extension register and accumulator)
PC	Program counter
P2	Pointer register 2
P3	Pointer register 3
S	Status register
SP	Stack pointer
T	Temporary register

4.2 ASSEMBLER CODING CONVENTIONS

Assembly language programs of source statements, each of which occupies one line of text. There are five types of source statements accepted by the assembler: the instruction, the comment, the pseudo instruction, the assignment, and the directive. Except for comment statements, each type of source statement contains from one to five fields in the following order.

[label field] operation field [operand field] [comment field] [identification field]

A sample coding form with the five fields delineated is shown in figure 4-2. Since the assembler accepts free-form statements, the user may disregard field boundaries. However, for clarity and readability, the use of field boundaries is highly recommended. Figure 4-3 shows an example of assembled code.

4.2.1 Label Field

The label field is optional and may contain a symbol used to identify the current statement when referenced in other statements. More than one label may appear in the label field, in which case any of the labels may be used to reference the labeled location. A label may appear by itself in a statement, in which case it refers to the next instruction or data word in the source program.

Example:

```
LABEL1:
LABEL2:
LABEL3: LABEL4: LD A, =1
```

4.2.2 Operation Field

The operation field is mandatory and contains a mnemonic operation code (opcode) that defines an assembler (such as a directive) or a machine operation (such as a load). A blank terminates the operation field.

Operation mnemonics are used in directive and instruction statements. Instruction statements define the machine operations necessary to perform the desired function. Valid operation codes for instruction statements are listed in appendix A. Directive statements control the process of program assembly and may generate data. Valid directive mnemonics are listed in appendix C.

4.2.3 Operand Field

The operand field contains entries that identify data to be acted upon by the statement. A space is not required to terminate the field. An operand entry is composed of one or more terms which represent a value. The value may be inherent in the term, in which case the term is self-defining (4.1.3.1); or the value may be assigned by the assembler program during assembly, in which case the term is symbolic (4.1.3.2). An arithmetic combination of terms is reduced to a single value by the assembler program as described in 4.1.4. The relationship of terms is shown in figure 4-1.

4.2.4 Comment Field

Comments are optional descriptive notes printed on the program listing for programmer reference. Comments should be included throughout the program to explain subroutine linkages, assumptions made, formats of inputs processed, and so forth. A comment may follow a statement on the same line, or the comment may be entered on one or more separate statement lines. The comment has no effect on the assembled program, but it is printed on the program listing.

The following conventions apply to comments:

1. A comment must be preceded by a semicolon (;).

2. All valid characters, including blanks, may be used in comments.

3. Comments should not extend beyond column 72, but a comment may be carried over on the following line (preceded by a semicolon).

CODING FORM

PROGRAM	PUNCHING INSTRUCTIONS	GRAPHIC									PAGE OF
PROGRAMMER		PUNCH									DATE

STATEMENT

LABEL 1 2 3 4 5 6 7 8	OPERATION 9 10 11 12 13 14 15 16	OPERAND 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	COMMENTS 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	IDENTIFICATION 73 74 75 76 77 78 79 80
1 2 3 4 5 6 7 8	9 10 11 12 13 14 15 16	17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

Figure 4–2. Sample Coding Form

```
                                .PAGE

;               PRINT NUMBER (from stack)

0186    83FE    PRNUM:  LD      EA,-2,P3        ; Check sign of number
0188    01              XCH     A,E
0189    6409            BP      PRN0
018B    01              XCH     A,E             ; Negative: replace with
018C                    NEGATE                  ;   absolute value
018D    8BFE            ST      EA,-2,P3
018F    C400            LD      A,='-'          ; Print Minus Sign
0191                    OUTC
0192    7403            BRA     PRN1
0194    C420    PRN0:   LD      A,=020          ; Positive: Print space
0196                    OUTC
0197    C400    PRN1:   LD      A,=0            ; Prepare to extract digits
0199    CDCC            ST      A,CHRNUM
019B    87FE            LD      EA,@ -2,P3      ; Reload the number
019D    23C5FF          PLI     P3,=TEMP        ; TEMP...TEMP+4 used
                                                ;   to hold digits
01A0    A40A00  PRN2:   LD      T,=10           ; Divide by 10
01A3    8DCA            ST      EA,DTEMP        ; Save dividend
01A5    0D              DIV     EA,T
01A6    08              PUSH    EA              ; Save quotient
01A7    A40A00          LD      T,=10           ; Find remainder
01AA    2C              MPY     EA,T
01AB    0B              LD      EA,T
01AC                    NEGATE
01AD    B5CA            ADD     EA,DTEMP
01AF    CF01            ST      A,@ 1,P3        ; Save digit (remainder)
01B1    95CC            ILD     A,CHRNUM        ; Increment digit count
01B3    3A              POP     EA
01B4    7CEA            BNZ     PRN2            ; Quotient=0 ?
01B6    01              XCH     A,E             ;   No - not done
01B7    6C03            BZ      PRN3
01B9    01              XCH     A,E
01BA    74E4            BRA     PRN2
01BC    C7FF    PRN3:   LD      A,@ -1,P3       ; Print the digits
01BE    F400            ADD     A,='0'          ; Most significant 1st
01C0                    OUTC
01C1    9DCC            DLD     A,CHRNUM
01C3    7CF7            BNZ     PRN3
01C5    C400            LD      A,=' '          ; Done: Print trailing
01C7                    OUTC                    ;   blank
01C8    5F              POP     P3
01C9    5C              RET
```

Labels Operation Field Operand Field Comments

Object Program

Hexadecimal memory address in which the object code is stored.

NOTE: Instructions without assembled data shown are macro calls.

Figure 4-3. Sample Program Listing

4.2.5 Identification Sequence Field

The identification sequence field is an optional entry that specifies program identification and/or statement sequence characters. If the field or a portion of the field is used for program identification, the identification is punched in the statement cards and is printed on the program listing. This field generally is not used with paper tape input.

As an aid to keeping source statements in order, the programmer may code a sequence of characters in ascending order in the identification sequence field.

The identification sequence field is fixed in columns 73 through 80 of the source image. Columns 73 through 80 are ignored by the assembler but are printed in the program listing.

In some cases, the width of the carriage of the printer is not wide enough to include the identification sequence field in the program listing.

4.2.6 Example of Statement

An example of an assembler statement follows:

```
Label      Operation  Operand    Comment

GETVAL:    LD         A, 1, P2   ;GET A VALUE
```

The label, GETVAL, is a symbolic name for the address of this instruction. Thus, GETVAL can be used in other statements (preceding or following) to address this statement. The mnemonic operation code, LD, stipulates the type of operation. The operand field specifies a register, A, a pointer, P2, and a displacement, +1; and the comment field contains a note that may be used by the programmer to identify quickly the action defined by the statement. See chapter 5 for other statement examples.

4.2.7 Delimiters

Certain characters have special meaning to the assembler in that they function as delimiters. Delimiters define the end of a source statement, a field, or a component of a field. The following list defines the delimiters recognized by the assembler.

Character(s)	Meaning	Use
blank	one or more blanks	Field separator or symbol terminator
,	comma	Operand separator, including macro parameters
'...'	pair of single quote characters	Character string delimiter
(...)	pair of parentheses	Subexpression delimiter
;	Semicolon	Comment field delimiter
:	Colon	Delimiter for symbols used as labels

Chapter 5

ASSEMBLER-DEPENDENT STATEMENTS

Assembler directives are statements that direct the assembler to perform some particular operation. They may or may not generate any machine code. Assembler directives provide the assembler with the following three types of information:

- Values of symbols
- How memory is to be mapped
- Assembly listing print options and formatting

5.1 ASSIGNMENT STATEMENT

The assignment statement assigns the value of the expression on the right side of the equals sign to the symbol on the left side of the equals sign.

Example:

```
START      = 0100      ;Starting address
```

The assignment statement may be preceded by a series of labels.

Example:

```
START1:
BEGIN:     FIRST    = 02D0
```

The assignment statement may set the location counter or refer to the current value of the location counter in an expression. The location counter assigns addresses to program statements at assembly time. It is similar to the Program Counter, which contains the address of the last byte of the current instruction at execution time. As each instruction or data area is assembled, the location counter always points to the next available location in memory.

The period "." is a special symbol used to specify the location counter. The location-counter symbol may appear on either side of the equals sign. If it appears on the left, it is assigned the value on the right side of the equals sign. The programmer may refer to the current setting of the location counter by referencing the"." in the expression to the right of the equals sign. Assignment statements using the location-counter symbol are coded as free-form statements. For example:

```
         .=20        ;SET LOCATION COUNTER TO 20
TABLE:   .=.+10      ;RESERVE 10 LOCATIONS FOR TABLE
```

If the "." appears on the left, the expression on the right must be defined during the first pass so subsequent label assignments may be made.

If the symbol on the left is not ".", then the expression on the right need not have a value during the first pass, but the expression must have a value during the second pass. This permits only one level of forward referencing. An example of more than one level of forward referencing follows:

```
FST:     A=B+2     This expression undefined during pass 2.
SND:     B=C-1     This expression undefined during pass 1.
THD:     C=25      This expression absolute
```

5.2 DIRECTIVE STATEMENTS

The directive statements control the assembly process and may generate data in the object program. The directive operator may be preceded by one or more labels, and may be followed by a comment. It occupies the operator field and is followed by either no operand or as many operands as are required by the particular operator.

Assembler directive operators and their functions are summarized in table 5-1. Note that all directive operators begin with a period for easy visual differentiation from the instruction operator mnemonics in the output listing. Each directive operator is described in more detail in the following paragraphs.

Table 5-1. Summary of Assembler Directives

Directive	Function
.TITLE	Identification of program.
.END	Physical end of source program.
.INCLD	Include separate file in assembler input.
.LIST	Listing output control.
.SPACE	Space n lines in output listing.
.PAGE	Output Listing to top-of-form.
.BYTE	8-bit (single-byte) data generation.
.DBYTE	16-bit (double-byte) data generation.
.ASCII	Data generation for character strings.
.LOCAL	Establish a new local symbol region.
.IF .ELSE .ENDIF	Conditional Assembly
.ADDR	Address constant generation.
.SET	Assign values to variables.
.STVAL	Access symbol table information for symbol.
.MACRO	Begin macro definition.*
.ENDM	End macro definition.*
.MLOC	Macro Local Symbols.*
.DO	Begin Macro-time looping.*
.ENDDO	End Macro-time looping.*
.EXIT	Exit Do loop.*
.IFC	Conditional Assembly.**
.ERROR	Macro error message generation.**
.MDEL	Macro delete.**

* Used only in macro definitions; see chapter 6 for descriptions.
** Macro related directive; see chapter 6 for descriptions.

5.2.1 .ASCII Directive

[label:] .ASCII string [,string...] [;comments]

The .ASCII directive stores data in successive memory locations by translating the characters in the string into their 7-bit ASCII equivalent code. Each string must be enclosed in single quote marks (‘). Each character occupies one byte in memory. The .ASCII directive is used primarily to generate messages for output on teletypewriter or printer. A single quote in the string must be preceded by a second quote mark.

Example:

```
.ASCII    'INPUT OF VALUE OF X'
```

5.2.2 .ADDR Directive

[label:] .ADDR expression [,expression...]

The .ADDR directive evaluates expression, decrements value by 1, and puts least-significant 8 bits in first byte and most-significant 8 bits in second byte. The .ADDR directive is used primarily to create data words which will be used for indirect or indexed addressing.

Examples:

1. Without a label

```
        .ADDR    THERE
```

2. With a label

```
LOC:    .ADDR    THERE
```

NOTE: LOC is assigned the value of the location of the first byte (least-significant byte) of the address value (THERE-1). LOC + 1 equals the value of the location of the second (most-significant) byte of the address value.

5.2.3 .BYTE Directive

[label:] .BYTE expression [,expression...] [;comments]

The .BYTE directive stores consecutively in memory one 8-bit byte for each given expression. If the directive has a label, if refers to the address of the first expression. The value of each expression must be in the range, − 128 to + 127 for signed data or 0 to 255 for unsigned data.

Examples:

1. Single expression without a label

```
        .BYTE    X'FF
```

2. Multiple expressions with a label

```
TBL:  .BYTE MPR +10,X'FF,X'00
```

NOTE: TBL is assigned the location counter value of the byte containing the expression MPR + 10.

5.2.4 .DBYTE Directive

```
[label:]    .DBYTE    expression [,expression...]    [;comments]
```

The .DBYTE directive stores 16-bit data in two consecutive 8-bit memory locations. Each expression of a .DBYTE directive is evaluated, and its value is placed in the next available pair of memory locations. The value of each expression must be in the range, −32768 to +32767 for signed data or 0 to 65535 for unsigned data.

The most-significant 8 bits are in byte 2. The least-significant 8 bits are in byte 1. If the directive has a label, it refers to the memory address of the first byte generated by the directive.

Examples:

1. Without a label

```
            .DBYTE    X'77FF
```

2. With a label

```
LABL:       .DBYTE    X'77FF
```

NOTE: LABL is assigned the value of the location of X'FF and LABL + 1 equals the value of the location of X'77.

A single character string appearing in a .DBYTE generates an 8-bit right-justified constant.

5.2.5 Macro-time Looping Directives

```
[label:] .DO        count        [;comments]
[label:] .ENDDO                  [;comments]
```

Macro-time looping is facilitated through the .DO and .ENDDO directives. These directives are used to delimit a block of statements which are repeatedly assembled. The number of times the block will be assembled is specified on the .DO directive. Following is the format of a .DO – .ENDDO block:

```
.DO        count
.
.
.
source
.
.
.
.ENDDO
```

NOTE: .DO, .ENDDO, and .EXIT are defined only within a macro definition.

5.2.6 Conditional Assembly Directives

```
[label:]    .IF       expression    [;comments]
            .ELSE                   [;comments]
            .ENDIF                  [;comments]
```

The conditional assembly directives selectively assemble portions of a source program based on the value of the expression in the .IF directive statement.

All source statements between a .IF directive and its associated .ENDIF are defined as a .IF – .ENDIF block. These blocks can be nested to a depth of ten. The .ELSE directive can be included optionally in a .IF – .ENDIF block. The .ELSE directive segments the block into two parts. The first part of the source statement block is assembled if the .IF expression is greater than zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF expression is greater than zero. If an error is detected in the expression, the assembler assumes a true value (greater than zero).

Examples:

1. Two-part conditional assembly

```
.IF        COMPR
.                    } Assembled if COMPR greater than zero
.
.
.ELSE
.                    } Assembled if COMPR less than or equal to zero
.
.
.ENDIF
```

2. Nested .IF – ENDIF block conditional assembly

```
                .IF        SMT
                .
                .        } Assembled if SMT is greater than 0
                .
                .ELSE
                .
                .        }
                .
                .IF        OBR
Assembled if    .
OBR is greater  .        } Assembled if SMT is less than
than zero and   .          or equal to zero
SMT is less than
or equal to zero
                .ENDIF
                .
                .        }
                .
                .ENDIF
```

Labels appearing on .IF statements are assigned the address of the next assembled instruction. Labels cannot be used on .ELSE or .ENDIF statements.

Listing of unassembled code may be controlled by appropriate use of .LIST directives (see paragraph 5.2.12)

5.2.7 Error Directive

[label:] .ERROR [string] [;comments]

The .ERROR directive generates an error message and an assembly error that is included in the error count at the end of the program. The directive is useful for parameter checking in macros.

Example:

```
;       ORG: SET ADDR ORIGIN AND TEST FOR OVERLAP
;
                .MACRO      ORG
                .IF         .>#1
                .ERROR      'PREVIOUS CODE OVERLAPS NEW ORIGIN'
                .ENDIF
                .=#1
                .ENDM
```

5.2.8 Exit Directive

[label:] .EXIT [;comments]

Early termination of looping in a .DO – .ENDDO block can be effected with the .EXIT directive.

This directive allows the current loop to finish and then terminates looping.

5.2.9 End Directive

[label:] .END [address] [;comments]

The .END directive signifies the physical end of the source program.

The optional address in the operand field, which may be either a symbol, a constant or an expression, indicates the address of the instruction in the program where you want program execution to begin. This address is the program entry point address, not the load point address. The program entry point address may be the same address as the load point but usually it is not.

If an address is not specified, the assembler assumes a default address of 0.

The .END directive only makes the entry point address available to the loader. Whether or not the address can be used is dependent on the loader.

Examples:

1. No entry point specified.

```
LAST:           .END
```

2. Jump to the entry point at 0A0.

```
                .END        0A0
```

3. Jump to the entry point labeled START.

```
                .END        START
```

Loader dependent option

5.2.10 Conditional Assembly Directive

[label:] .IFC string1 operator string2 [;comments]

The .IFC directive allows conditional assembly based on character strings rather than the value of an expression as in the .IF directive. String1 and String2 are the character strings to be compared. Operator is the relational operator between the strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFC are assembled until a .ELSE or a .ENDIF is encountered. The .ELSE and .ENDIF directives have the same effect with the .IFC directive as they do with the .IF directive.

The primary application of the .IFC is to compare a parameter value such as #3 against a specific string. For example:

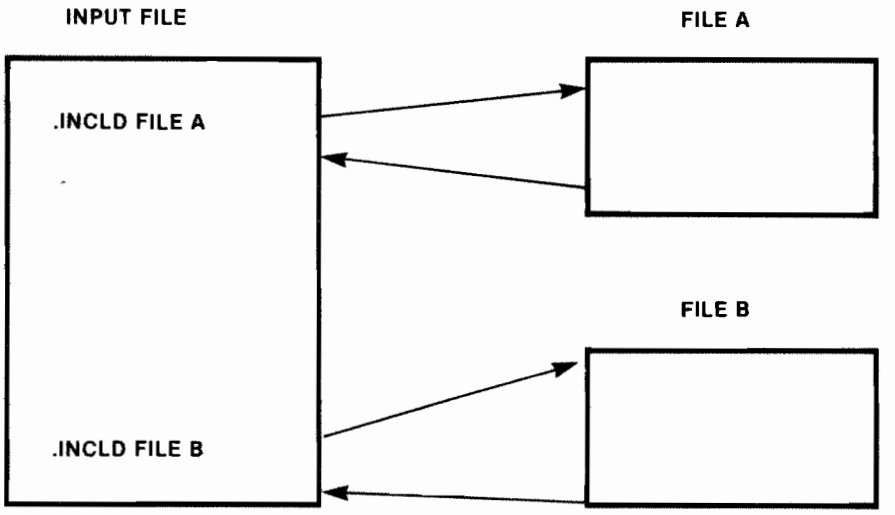
.IFC' #3' NE' INTEGER'

5.2.11 Include Directive

[label:] .INCLD filename [;comments]

The .INCLD directive enables a user to include one or more additional source files as part of the input stream to the assembler. Only source files residing on diskette can be included.

When a .INCLD directive is encountered during an assembly, input to the assembler is switched from the main source file to the included source file. The input is switched back to the main source file when the end-of-file mark (EOF) of the included file is reached. (See the figure below.)



The .INCLD directive can be used anywhere in the main source file but nesting of .INCLD is prohibited (i.e., there cannot be a .INCLD in the included file). An error in the use of the .INCLD such as included file not found, disk unavailable, or including a file of illegal file type, will cause abortion of the assembly.

5.2.12 List Directive

[label:] .LIST immediate [;comments]

The .LIST directive controls the program listing.

Control of the various list options depends upon the state of the five least-significant bits of the evaluated expression in the operand field. Table 5-2 gives the list options available in the order of their priority. The options are usually combined (ORed) to give the desired type of listing.

Table 5-2. List Options

Control Function	Bit Positions	Bit Binary Value	Six-bit Hexa-decimal Value	Description
Master List	0	0	00	Suppress all listing
		1	01	*Full listing
.IF List	1	0	00	*Suppress listing of un-assembled code
		1	02	Full listing (of .IF s and .IFC's)
Macro List	2,3	00	00	*List only macro calls
		10	08	List only code generated by macro calls
		11	0C	List all code expanded during macros calls
Binary List	4	0	00	List only the first two bytes of generated data
		1	10	*List all the binary output by statements generating more than one word (e.g., .ASCII)
Include List	5	0	00	*List only error lines for the included file
		1	20	List the included file (source statements from the included files are listed without line numbers)

*indicates default

Options are usually combined to give the desired type of listing. Some examples follow:

1. Full listing:

```
.LIST  01
```

2. Full listing and list all code expanded during macro calls.

```
.LIST  0D          Full list
  or
.LIST  01  !  0C   list all code expanded during macro calls
```

3. Suppress listing.

```
.LIST  0
```

5.2.13 Local Directive

```
[label:]      .LOCAL      [;comments]
```

The .LOCAL directive establishes a new program section for local symbols (symbols beginning with a dollar sign ($)). Designated symbols between two .LOCAL directive statements have the value assigned to them only within that particular section of the program. Note that a .LOCAL directive is assumed at the beginning and the end of a program; thus, one .LOCAL directive within a program divides the program into two sections.

If the first character of a symbol is a dollar sign ($), the assembler replaces the "$" in the symbol name by the local region number, beginning with 0. Each time a .LOCAL directive is encountered, the number of the local region (0-255) is advanced by one. Therefore, up to 255 .LOCAL directives can appear in one assembly.

Example:

```
.LOCAL
```

5.2.14 Macro Definition Directives

```
[label:]   .MACRO   mname [parameters]   [;comments]
[label:]   .ENDM                         [;comments]
```

The following form is used to define a macro:

```
.MACRO mname      [parameters]

        .
        .
        .

        macro body

        .
        .
        .

.ENDM
```

where:

(1) .MACRO is the directive that initiates the macro definition. Macros must be defined before their use. It is legal to define a macro with the same name as an already existing macro. The latest definition is always the operative one, but previous definitions are not discarded. They may be reactivated by using the .MDEL directive to delete the last macro definition (see 5.2.15).

(2) "mname" is the name of the macro (the name used to "call" the macro). The macro name must adhere to all rules for symbols (see 4.1.3.2).

(3) Parameters is the optional list of parameters used in the macro definition. The list of parameters must adhere to all the rules for expressions (see 4.1.4).

The following are examples of legal and illegal .MACRO directives:

Legal		Illegal		Reason Illegal
.MACRO	MAC,A,B	.MACRO	SUB,1	Special character is used in parameter.
.MACRO	$ADD,OP1,OP2	.MACRO	1MAC,C,D	First character in macro name is illegal.
.MACRO	LIST,$1	.MACRO	MACB,25	First character in parameter must be alphabetic or $.
.MACRO	MSG3	.MACRO	M$AC	Special character is used in macro name.

(4) Macro body consists of assembly language statements. The macro body may contain simple text (see below), text with parameters (see 6.5) and macro-time operators (see 6.5.3 and 6.5.4).

(5) .ENDM is the directive that terminates the macro definition.

5.2.15 Delete Macro Directive

[label:] .MDEL mname [,mname...] [;comments]

The .MDEL directive deletes macro definitions from the macro definition table and frees the buffer space used by the definitions.

Example:

```
.MDEL    DELAY,MAC1
```

5.2.16 Macro Local Label Directive

[label:] .MLOC symbol [,symbol...] [;comments]

When a label is defined within a macro, a duplicate definition results with the second and each subsequent call. The problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Local symbols are replaced with unique names at expansion time with ZZxxxx, where xxxx is a 4-digit hexadecimal number. The user should avoid using his own labels of the above form as it may cause duplicate definition errors. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbols it declares local. It it does not, no error will be reported, but symbols used before the .MLOC will not be recognized as local.

5.2.17 .PAGE Directive

[label:] .PAGE [string] [;comments]

The .PAGE directive spaces forward to the top of the next page on the output listing. The optional string is printed as a page title on each page until a .PAGE directive containing a new string is encountered. No action is taken (except for a new page title) if the .PAGE directive is encountered immediately after an assembler generated top-of-page request.

Example:

```
.PAGE   'TTY I/O ROUTINES'
```

5.2.18 .SET Directive

[label:] .SET symbol,expression [;comments]

The .SET directive is used to assign values to reassignable (set) variables. A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times.

Examples:

```
.SET    A,100          ;SET A = 100
.SET    B,50           ;SET B = 50
.SET    C,A-25* (B/4)  ;SET C = A - 25* (B/4)
```

NOTE: This expression is always evaluated from left to right regardless of the operators used between the variables and constants. Parentheses may be used to group subexpressions.

The .SET directive is used to assign values to symbols (variables). A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times. Set variables are useful during macro expansion to control macro-time looping and macro communication. To insure value correspondence between pass one and pass two of the assembler, all values in the expression must be defined before use in a .SET directive. If a value is not previously defined, an error is reported and a value of zero is returned.

5.2.19 .SPACE Directive

[label:] .SPACE immediate [;comments]

The .SPACE directive skips forward a specified number of lines on the output listing.

Examples:

1. Skip 20 (decimal) lines

```
.SPACE 20
```

2. Skip 20 (hexadecimal) lines

```
.SPACE 020
  or
.SPACE X'20
```

5.2.20 Symbol Table Value Directive

[label:] .STVAL symbol1, symbol2 [;comments]

The .STVAL directive retrieves symbol table information for symbol 2, structures the information into a 16-bit value and assigns that value to symbol 1.

All the necessary information is contained in the six least-significant bits of the value assigned to symbol1. It can be interpreted as follows:

Bit	Function
0	Defined bit (1 if symbol 2 has been assigned a value)
1	Previously defined bit (1 if symbol 2 is defined previously in this pass)
2	Long symbol bit (1 if symbol 2 is made up of more than four characters)
3	.SET bit (1 if the value of symbol 2 can be reassigned)
4	Used bit (1 if symbol 2 has been used somewhere else in the program)
5	Local bit (1 if symbol 2 is a local symbol)

If symbol 2 is not in the symbol table, symbol 1 assumes the value of −1 (0FFFF).

5.2.21 Title Directive

[label:] .TITLE symbol [,string] [;comments]

The .TITLE directive identifies the load module in which it appears with a symbolic name and an optional definitive title. If a .TITLE directive does not appear in the program, the load module is given the name MAINPR. If more than one .TITLE directive is used, the last one encountered specifies the symbolic name.

The symbolic name and the string must meet the symbol and string construction restrictions discussed in chapter 4.

Example:

```
.TITLE    TBLKP,'TABLE LOOKUP'
```

The string, if specified, will be printed at the top of each listing page and has a maximum length of 22 characters.

Chapter 6

MACROS

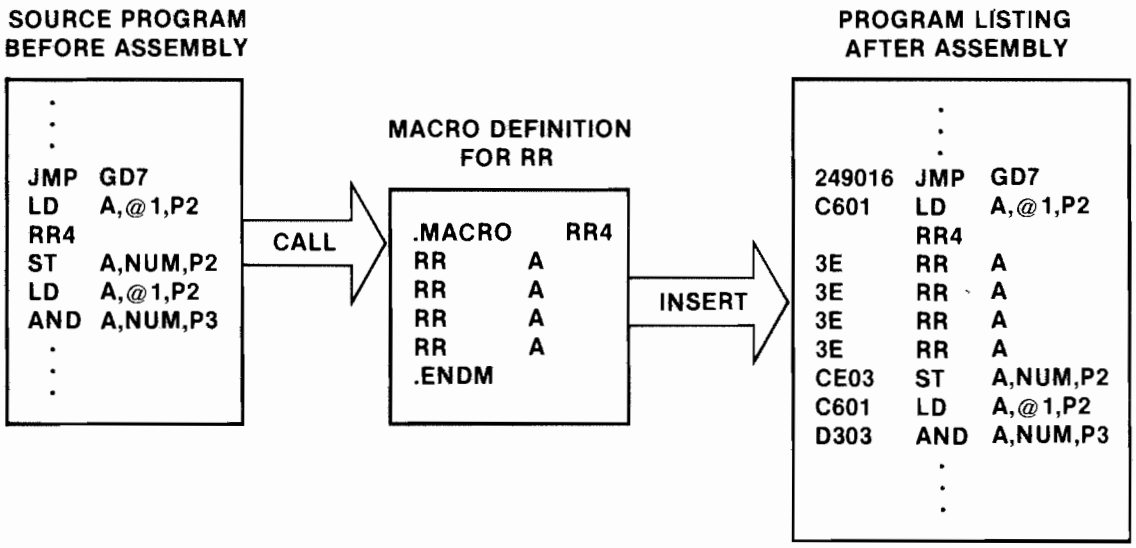
6.1 INTRODUCTION

Programming in simple assembly language enables a user to be as efficient with his microprocessor resources as his capabilities allow. With assembly language, the user can specify explicitly every detail of the program operation: indeed, he must specify every detail. Because of this, a program in assembly language often takes longer to write than the same program written in a high-level language, which fills in many details automatically according to its internal design. This design may or may not be compatible with either the language of the machine on which the high-level language operates or the user's problem. Ideally, the user would like a programming language that (1) most closely resembles the machine language, when it is appropriate, and (2) is in a form most suitable (natural) for expressing the user's problem. The language should fill in details whenever they are routine and should leave the user free to specify the details whenever they are crucial. This ideal can often be closely approximated by the use of a versatile programming tool known as macros.

Macros are a form of text replacement that provide an automatic code-generation capability completely under the user's control. With macros, a user can gradually build a library tailored to his application, and, with a library of macros oriented toward a particular application, a user who is not a software expert can produce efficient machine-language code; and an experienced user can significantly reduce his program development time.

6.2 BASIC MACRO CONCEPTS

The main use of macros is to insert assembly language statements into a source program, as shown in figure 6-1. In the example, the original source program contains a macro instruction, or macro call, named RR4. RR4 is a macro that rotates the Accumulator right four times. When the assembler processes RR4, it inserts the predefined sequence of assembly language from the macro definition named RR4 into the source program immediately after the point of call (RR4). The process of inserting the text of the macro definition into the source program is called macro expansion. The expanded macro then is processed as if it were part of the original source program. You will note that the macro call itself does not produce any machine language code. The directives used to define the limits of the macro definition (.MACRO and .ENDM) are explained in detail later in this chapter.



The above example illustrates three aspects of a macro: the definition, the reference, and the expansion.

Figure 6 – 1. Statement Insertion

6.3 DEFINING A MACRO

Defining a macro means preparing the statements that constitute a macro definition. To define a macro, the following must be done:

- Give it a name.
- Declare any parameters to be used.
- Write the statements it contains.
- Establish its boundaries.

The following form is used to define a macro:

```
.MACRO      mname      [parameters]
.
.
.
macro body
.
.
.
.ENDM
```

where:

(1) .MACRO is the directive that initiates the macro definition. Macros must be defined before their use. It is legal to define a macro with the same name as an already existing macro. The latest definition is always the operative one, but previous definitions are not discarded. They may be reactivated by using the .MDEL directive to delete the last macro definition (see 6.12).

(2) "mname" is the name of the macro (the name used to "call" the macro). The macro name must adhere to all rules for symbols (see 4.1.3.2).

(3) Parameters is the optional list of parameters used in the macro definition. The list of parameters must adhere to all the rules for expressions (see 4.1.4).

The following are examples of legal and illegal .MACRO directives:

Legal		Illegal		Reason Illegal
.MACRO	MAC,A,B	.MACRO	SUB,1	Special character is used in parameter.
.MACRO	$ADD,OP1,OP2	.MACRO	1MAC,C,D	First character in macro name is illegal.
.MACRO	LIST,$1	.MACRO	MACB,25	First character in parameter must be alphabetic or $.
.MACRO	MSG3	.MACRO	M$AC	Special character is used in macro name.

(4) Macro body consists of assembly language statements. The macro body may contain simple text (see below), text with parameters (see 6.5) and macro-time operators (see 6.5.3).

(5) .ENDM is the directive that terminates the macro definition.

The simplest form of a macro definition is one with no parameters or macro operators. The macro body is simply a sequence of assembly language statements which are substituted for each macro call. The following show several examples of simple macro definitions:

```
;RESET STACK POINTER
.MACRO      SSTKP                    Begin the macro definition
LD          EA, =STRT  ┐
XCH         EA,SP      ├ Macro body
ST          EA,STKP    ┘
.ENDM                                End the macro definition

;RESTORE STACK POINTER
.MACRO      RSTKP
LD          EA,STKP
XCH         EA,SP
.ENDM

;DELAY ONE MILLISECOND
LD          A, =57
ST          A,CNT
DLD         CNT
BNZ         . -2
.ENDM
```

6.4 CALLING A MACRO

Once a macro has been defined, it then may be called. A macro is called by placing the macro name in the operation field of an assembly language statement, and the parameters in the operand field. The following form is used for a macro call:

```
mname      [parameters]
```

where:

(1) "mname" is the name previously assigned in the macro definition.

(2) "parameters" is the list of input parameters. When a macro is defined without parameters, the parameter list is omitted from the call.

A call to the delay macro, defined in 6.3, would be as follows:

```
Source Program                                   Program Listing
Before Assembly                                  After Assembly

.                                                     .
.                                                     .
.                                                     .
                                                      .
RSTKP  ═══ generates ═══>                        RSTKP
.
.                                            31FC LD      EA,STKP
.                                            4D   XCH     EA,SP
                                                      .
                                                      .
                                                      .
                                                      .
```

6.5 USING PARAMETERS

The power of a macro can be increased tremendously through the use of the optional parameters. The parameters allow variable values to be declared when the macro is called.

6.5.1 Macro Definition

The delay macro previously illustrated could be made more powerful through the use of parameters. By making the delay constants depend on the call, the same macro can be used for a variety of delays. In this example, UCYCS is a formal parameter and can be replaced with any value at expansion time.

```
;DELAY UCYCS MICROCYCLES
.MACRO      DELAY2,UCYCS
LD          A, = (UCYCS-14)
ST          A, CNT
DLD         A, CNT
BNZ         .-2
.ENDM
```

Parameters need not be variables or numeric values, but can be any string. The following macro, for example, takes an ASCII string as input and generates a message string in memory suitable for input to the MESG routine.

```
            .MACRO      MSGSTR,LABEL,STRING
LABEL:      .ASCII      'STRING'
            .BYTE       0
            .ENDM
```

The following macro generates a call to the MESG routine with the name of the message as input.

```
            .MACRO      MESG,MSGNAM
            JSR         07EA7
            .DBYTE      MSGNAM
            .ENDM
```

Note the principle here: the hexadecimal firmware address is maintained centrally in the MESG macro, not scattered all over the code as the macro calls will be.

Finally, here are some macros of obvious utility in a 70-Series environment:

```
; MULTIPLY
.MACRO      MULT,M1,PX,M2,PY
LD          EA,M1,PX
LD          T,M2,PY
MPY         EA,T
.ENDM

; DIVIDE
.MACRO      DIVIDE,M1,PX,M2,PY
LD          EA,M1,PX
LD          T,M2,PY
DIV         EA,T
.ENDM
```

6.5.2 Calling a Macro With Parameters

When parameters are included in a macro call, the following rules apply to the parameter list:

1. Commas or blanks delimit parameters.
2. Consecutive blanks are treated as a single delimiter.
3. A comma leading, following, or imbedded in a string of blanks is treated as a single delimiter.
4. A semicolon terminates the list and starts the comment field.
5. Quotes may be included as part of a parameter except as the first character of a parameter.
6. A parameter may be enclosed in quotes (') in which case the quotes are removed and the string is used as the parameter. This function is useful when blanks, commas, or semicolons are to be included in the parameter.
7. To include a quote in a quoted parameter, it must be entered as two consecutive quotes.
8. Missing or null parameters are treated as strings of length zero.

The following examples illustrate these rules:

NOTE: In these examples, for clarity, # indicates the parameter; strings are delimited by double primes (″).

Macro Call	Parameter List		
MAC1 NAME NO ONE MAN	#1	″NAME″	
	#2	″NO″	
	#3	″ONE″	
	#4	″MAN″	
	#5, #6, . . .		(NULL PARAMETER)
MAC2 MADAM 'I' 'M' ADAM	#1	″MADAM″	
	#2	″I'M″	
	#3	″ADAM″	
	#4, #5, . . .	″ ″	(NULL PARAMETER)
MAC3 'A MAN','A PLAN','A CANAL';PANAMA			
	#1	″A MAN″	
	#2	″A PLAN″	
	#3	″A CANAL″	
	#4, #5, . . .	″ ″	(NULL PARAMETER)
MAC4 POOR,,'IS IN A DROOP'	#1	″POOR″	
	#2	″ ″	(NULL PARAMETER)
	#3	″IS IN A DROOP″	
	#4, #5, . . .	″ ″	(NULL PARAMETER)

The following examples show some macro calls to macros defined in 6.3. These illustrate the parameter parsing and substitution.

Macro Call		Generated Code	
DELAY2 100		LD	A,(100-14)
		ST	A,CNT
		DLD	A,CNT
		BNZ	.-2
MSGSTR PROMPT 'ENTER VALUE'	PROMPT:	.ASCII	'ENTER VALUE'
		.BYTE	0

6.5.3 Parameters Referenced by Number

6.5.3.1 '#' — Number of Parameters

'#' is a macro operator that references the parameter list in the macro call. When used in an expression, it is replaced by the number of parameters in the macro call. The following .IF directive, for example, causes the conditional code to be expanded if there are more than 10 parameters in the macro call:

```
.IF     #>10
```

6.5.3.2 '#N' — Nth Parameter

When used in conjunction with a constant or variable, the '#' operator references individual parameters in the parameter list. The following example demonstrates how this function is used.

```
.MACRO    X
.BYTE     #1,#2,#3
.ENDM

X         3,5,2        generates    .BYTE    3,5,2
```

This relieves the need for naming each parameter in a long list and allows powerful macros to be defined using arbitrary numbers of parameters.

6.5.4 '∧' — Concatenation

The '∧' macro operator is used for concatenation. When found, the '∧' is removed from the output string and the strings on each side of the operator are compressed together after parameter substitution. If a set variable is used with the '∧' operator, it is converted to a hexadecimal number before being placed into the output stream.

Example:

```
.MACRO    IMAGINARY,X
R ∧ X:    .BYTE    0
I ∧ X:    .BYTE    0
.ENDM
```

Another example of the use of this operator is shown in 6.9.3.

6.6 LOCAL SYMBOLS

```
.MLOC     symbol[,symbol...]        [;comments]
```

When a label is defined within a macro, a duplicate definition results with the second and each subsequent call. The problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Local symbols are replaced with unique names at expansion time with ZZxxxx, where xxxx is a 4-digit hexadecimal number. The user should avoid using his own labels of the above form as it may cause duplicate definition errors. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbols it declares local. If it does not, no error will be reported, but symbols used before the .MLOC will not be recognized as local.

6.7 CONDITIONAL EXPANSION

The versatility and the power of the macro assembler is enhanced by the conditional assembly directives. The conditional assembly directives (.IF, .ELSE and .ENDIF) allow the user to generate different lines of code from the same macro simply by varying the parameter values used in the macro calls. Three relational operators are provided:

= (equal)
< (less than)
> (greater than)

Also two operators are provided for string comparisons:

EQ (equal)
NE (not equal)

6.7.1 .IF, .ELSE, .ENDIF Directives

When the macro assembler encounters a .IF directive within a macro expansion, it evaluates the relational operation that follows. If the expression is satisfied (evaluated greater than 0), the lines following the .IF are expanded until a .ELSE or a .ENDIF directive is encountered. If the expression is not satisfied (evaluated less than or equal to 0), only the lines from the .ELSE to the .ENDIF are expanded. See 5.2.6 for additional information on the conditional assembly directives.

Example:

```
;SHIFT THE CONTENTS OF RAM ADDRESS X
; RIGHT IF N > 0, LEFT OTHERWISE
.MACRO      SHIFT,X,N
LD          A,X
.IF         N > 0
SR          A
.ELSE
SL          A
.ENDIF
ST          A,X
.ENDM
```

6.7.2 .IFC Directive

[label] .IFC string1 operator string2 [;comments]

The .IFC directive allows conditional assembly based on character strings rather than the value of an expression as in the .IF directive. String1 and String2 are the character strings to be compared. Operator is the relational operator between the strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFC are assembled until a .ELSE or a .ENDIF is encountered. The .ELSE and .ENDIF directives have the same effect with the .IFC directive as they do with the .IF directive.

The primary application of the .IFC is to compare a parameter value such as #3 against a specific string. For example:

```
.IFC '#3' NE 'INTEGER'
```

6.8 USEFUL DIRECTIVES

6.8.1 .SET Directive

```
[label]     .SET     symbol,expression     [;comments]
```

The .SET directive is used to assign values to symbols (variables). A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times. Set variables are useful during macro expansion to control macro-time looping and macro communication. To insure value correspondence between pass one and pass two of the assembler, all values in the expression must be defined before use in a .SET directive. If a value is not previously defined, an error is reported and a value of zero is returned.

6.8.2 .MDEL Directive

```
[label]     .MDEL     mname[,mname...]     [;comments]
```

The .MDEL directive deletes macro definitions from the macro definition table and frees the buffer space used by the definitions.

Example:

```
.MDEL     DELAY,MAC1
```

6.8.3 .ERROR Directive

```
[label]     .ERROR     [string]     [;comments]
```

The .ERROR directive generates an error message and an assembly error that is included in the error count at the end of the program. The directive is useful for parameter checking in macros.

```
;LOAD POINTER (OTHER THAN PC) IMMEDIATE
.MACRO     LPI,PTR,EXPR
.IF        PTR < 4
.IF        PTR > 0
.SET       SYMB,EXPR
LD         PTR,=SYMB
.ELSE
.IF        PTR=0
.ERROR     'LDI WILL NOT WORK WITH THE PC'
.ELSE
.ERROR     'ILLEGAL POINTER VALUE'
.ENDIF
.ENDIF
.ENDM
```

6.9 MACRO-TIME LOOPING

6.9.1 .DO and .ENDDO Directives

```
[label]     .DO      count     [;comments]
[label]     .ENDDO             [;comments]
```

Macro-time looping is facilitated through the .DO and .ENDDO directives. These directives are used to delimit a block of statements which are repeatedly assembled. The number of times the block will be assembled is specified on the .DO directive. Following is the format of a .DO – .ENDDO block:

```
.DO         count

.
.
.

source

.
.
.

.ENDDO
```

NOTE: .DO, .ENDDO, and .EXIT are defined only within a macro definition.

6.9.2 .EXIT Directive

```
[label]     .EXIT                   [;comments]
```

Early termination of looping in a .DO – .ENDDO block can be effected with the .EXIT directive.

This directive allows the current loop to finish and then terminates looping.

6.9.3 Examples of Macro-Time Loops

The following examples show the use of the .DO, .ENDDO, and .EXIT directives. The macro CTAB generates a constant table from 0 to MAX where MAX is a parameter of the macro call. Each word has a label DX: — where X is the value of the data word.

```
            .MACRO      CTAB,MAX
            .SET        X,0
            .DO         MAX+1
D ∧ X:      .BYTE       X
            .SET        X,X+1
            .ENDDO
            .ENDM
```

Now a call of the form:

```
            CTAB        10
```

generates code equivalent to:

```
            .SET        X,0
D00:        .BYTE       X
            .SET        X,X+1
D01:        .BYTE       X
            .SET        X,X+1
D02:        .BYTE       X

            .
            .
            .
```

```
        .SET    X,X+1
D09:    .BYTE   X
        .SET    X, X+1
D0A:    .BYTE   X
```

The macro MSGLST generates a call to routine MESG with each of the parameters on the macro call. The parameter list may be any length.

```
        .MACRO  MSGLST
        .SET    X,0
        .DO     -1          ;SET FOR INFINITE LOOPING
        .SET    X,X+1
        .IF     X > #       ;CHECK IF NUMBER OF TIMES THRU LOOP (X)
                            ;IS > NUMBER OF PARAMETERS CALLED BY
                            ;MSGLST
        .EXIT               ;YES
        .ELSE
        MESG    #X          ;NO, CALL MESG MACRO
        .ENDIF
        .ENDDO
        .ENDM
```

Now a call of the form:

```
        MSGLST  MSG1,MSG2
```

generates code equivalent to:

```
        JSR     07EA7
        .DBYTE  MSG1
        JSR     07EA7
        .DBYTE  MSG2
```

NOTE: Care must be taken when writing macros that generate a variable number of data words through the use of the .IF or the .DO directives. If the operands on these directives are forward references, their values change between pass 1 and pass 2 and the number of generated words may change. Should this be the case, all labels defined after the macro call that has changed values generate numerous assembly errors of the following form:

```
ERROR DUP. DEF.
```

6.10 NESTED MACRO CALLS

Nested macro calls are allowed, that is, a macro definition may contain a call to another macro. When a macro call is encountered during macro expansion, the state of the macro currently being expanded is saved and expansion begins on the nested macro. Upon completing expansion of the nested macro, expansion of the original macro continues. Depth of nesting allowed will depend on the parameter list sizes, but on the average about 8 levels of nesting will be allowed.

A logical extension of a nested macro call is a recursive macro call, that is a macro that calls itself. This is allowed, but care must be taken that an infinite loop is not generated.

The following macros illustrate the value of the nesting capability. The first macro is a utility macro used to check if the Extended Accumulator is referenced. If not, it is loaded from another register.

```
;CHECK      EA REFERENCE
.MACRO      CHKEA,REG
.IF         REG NE ''EA''
LD          EA,REG
.ENDIF
.ENDM
```

```
;SIMULATE BRANCH CONDITION
.MACRO      BOC,COND,REG,DEST
CHKEA       REG
.IFC COND   EQ "BP"
LD          A,E
.ELSE
OR          A,E
.ENDIF
COND        DEST
.ENDM
```

This will generate the following code:

```
.
.
.
.

BOC         BP,P2,XYZ
LD          EA,P2
LD          A,E
BP          XYZ

.
.
.
.
```

6.11 NESTED MACRO DEFINITIONS

A macro definition can be nested within another macro. Such a macro is not defined until the outer macro is expanded and the nested .MACRO statement is executed. This allows the creation of special-purpose macros based on the outer macros parameters and, when used with the .MDEL directive, allows a macro to be defined only within the range of the macro that uses it.

The following is an example in which a specific location, LOC, is divided by a constant.

```
.MACRO      DIVIDE,D1,LOC1,D2,LOC2,D3,LOC3
.SET        AAA,# - 1/2
.SET        Q,2
.SET        R,3
.DO         AAA
.MACRO      #Q,#R,CONST
PLI         P2, = #2
LD          T,(P2)
POP
LD          EA, = CONST
DIV         EA,T
.ENDM
.SET        Q,R + 1
.SET        R,Q + 1
.ENDDO
.ENDM
```

Chapter 7

PROGRAMMING TECHNIQUES

7.1 SCOPE

The intent of this section is to illustrate to the user some of the more useful techniques in speeding up or compressing his code.

7.2 SUBROUTINES

This is a primary technique for the compression of code. Its purpose is to replace a repeated string of code in a program with one statement, a 'CALL' to a subroutine in the Series-70 processor; this may be either the single-byte 'CALL' instruction or a JSR instruction.

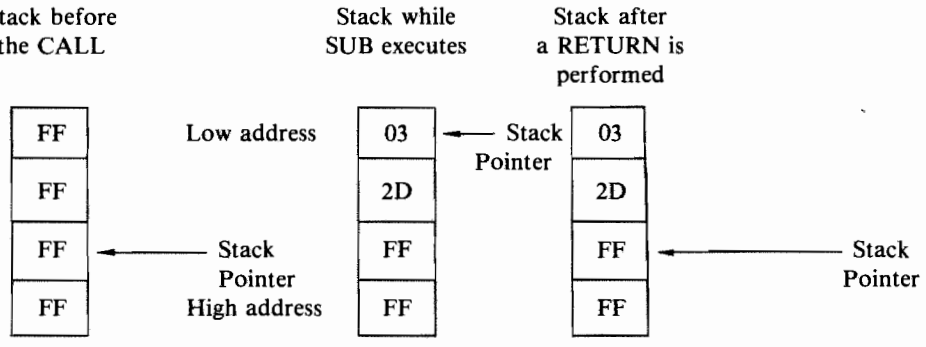
Example:

A subroutine "SUB" must be called.

```
2D00 20          JSR  SUB
2D01 00                   \
2D03 4C                    \ 4C00          SUB:  .
2D04                       /                     .
.                         /                      .
.                        /                       .
.                                                .

3018 20          JSR  SUB                        RET
3019 00
301A 4C
301B
.
.
.
.
```

When the first call is executed, address 02D03 is pushed onto the stack indicated by the stack pointer, and control is transferred to 04C00. Execution of the RETURN statement in SUB will cause the top entry to be popped off the stack into the program counter, causing execution to continue at 02D03 (since the CALL statement is three bytes long).



When the second call is executed, address 0301B is pushed onto the stack, and control is again transferred to SUB. This time, either RETURN instruction will cause execution to resume at 0301B.

Note that SUB could have called another subroutine during its execution, causing another address to be pushed onto the stack. This can occur as many times as necessary, limited only by the size of memory available for the stack.

Note also that any subroutine could push data onto the stack for storage without affecting the call and return sequences as long as the same amount of data is popped off the stack before executing a RETURN instruction.

So far the JSR instruction for subroutines has been discussed. In order to reduce the amount of code generated by the subroutine calls themselves, a "CALL" instruction is provided. This offers a saving of 2 bytes every time a subroutine is called subsequent to the first call.

Hence to compress the program use all the available 16 calls, starting with the most used subroutines.

A 'CALL' can be set up as follows:

```
                          . = 020
0020 00FF     SUB1P:      .ADDR SUB1
0022 01FF     SUB2P:      .ADDR SUB2

.             .           .
.             .           .
.             .           .

0100 010      CALL        0

.             .           .
.             .           .
.             .           .

0200 011      CALL        1

.             .           .
.             .           .
.             .           .
              etc.
```

Finally, whatever subroutine is called, bear in mind that multiple entry and exit points can also be put to good use.

For example, a routine may left justify a number and then check if the exponent is greater than a specific magnitude.

```
SHCHK:   LD     EA,MNT,P2    ;Get mantissa
LOOP:    XCH    A,E          ;Check if right justified
         BP     NO           ;JMP if not
         XCH    A,E          ;Restore E,A
         ST     EA,MNT       ;Store back mantissa
CHK:     LD     A,EXP        ;Get exponent
         SUB    A,LIM        ;Check against limit
         BP     OUT          ;Exit if greater
         POP    EA           ;Get return address
         ADD    EA,=0002     ;Increment x 2
         LD     PC,EA        ;Return to old ret + 2
OUT:     RET                 ;Exit
NO:      XCH    A,E          ;Restore E,A
         SL     A            ;Shift left
         LD     T,EA         ;Save EA
         ILD    A,EXP        ;Increment exponent
         LD     EA,T         ;Regain EA
         BRA    LOOP         ;Loop
```

The example shows two exit points to the normal return and to return + 2 as a conditional. In addition, the exponent check may be used as a subroutine on its own. Thus a call to CHK will merely compare the value stored in EXP and exit.

7.3 TRANSFERRING DATA TO SUBROUTINES

A subroutine often requires data to perform its operations. In the simplest case, this data may be transferred in one or more registers.

Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in registers P2 and P3.

The technique used to transfer data to and from a subroutine is an important factor in obtaining an efficient program. The most straight-forward method involves passing parameters in working registers (P2 and P3) and returning results the same way.

But in certain classes of subroutines (those whose parameters are not normally computed but are known at assembly time) this commonly used technique is not efficient. Consider a routine that must type a single ASCII character. If this character were transferred in the working registers (for example, A) then to type the letter 'X' the sequence would be coded thus:

```
LD      A, = 'X'     ;PRESET A
JSR     TYPE         ;TYPE IT
```

As an alternate, consider storing the parameter immediately following the CALL instruction. You can then eliminate LD and reduce each call by one byte:

```
JSR TYPE     ;CALL TYPE ROUTINE
.BYTE 'X'    ;PARAMETER
```

This approach resembles the method (discussed above) of passing parameters to restart routines. You must add a few extra bytes of code to the type routine to complete the interface:

```
TYPE:    POP     P2          ;Put return address into P2
         LD      A,@1,P2     ;Increment return address
         PUSH    P2          ;Save corrected return address
         LD      A,0,P2      ;Fetch parameter byte
TYPE2:   ;continue with routine
```

NOTE: Two loads are necessary due to post-increment auto-indexing and pre-incrementing of the Program Counter.

You can easily extend this method for passing parameters to more than one parameter, each of which could be longer than one byte (address, for example).

7.4 TEXT PROGRAMMING TECHNIQUES

When programs require extensive dialog, textual display, or printout, attention should be given to the technique that programs the textual printout, since it is likely to be subject to modification. One technique that readily lends itself to the 807X is the literal pool. A literal pool is an area within an assembled program where the literals used within the program are stored. Within the literal pool, all duplication of text is eliminated. For example, consider the following five messages:

1. ENTER 5 COEFFICIENTS
2. COEFFICIENT OUT OF ALLOWED RANGE. REENTER
3. ANSWER =
4. ANOTHER?
5. NO VALID ANSWER. REENTER 5 COEFFICIENTS

Text for the five messages may be stored in a literal pool as shown below.

```
2E52        L1:     .ASCII    '.RE-'
452D
454E        L2      .ASCII    'ENTER'
5445
5220
35          L3:     .ASCII    '5'
2C43        L4      .ASCII    'COEFFICIENT'
4F45
4646
4943
4945
4E54
53          L5:     .ASCII    'S'
204F        L6:     .ASCII    'OUT OF ALLOWED RANGE'
5554
204F
4620
414C
4C4F
5745
4420
5241
4E47
45
414E        L7:     .ASCII    'ANOTHER?'
4F54
4845
523F
414E        L8:     .ASCII    'ANSWER'
5357
4552
3D20        L9:     .ASCII    '='
4E4F        L10:    .ASCII    'NO VALID'
2056
414C
4944
20
            L11:    .=.+1
```

Messages are created by indexing the literal pool using a 2-byte repeating sequence. Byte 1 holds the displacement from the base of the literal pool to the first required character. Byte 2 holds the value of the number of characters to be printed. If the first byte pair holds the value X'FF, it signifies the end of the message; otherwise, another segment of the message is sought in the next byte pair. Each of the five messages described above could be created by the index sequence shown below.

```
04      I1:     .BYTE  L2-L1     ;ENTER 5 COEFFICIENTS
14              .BYTE  L6-L2
FF              .BYTE  X'FF
0B      I2:     .BYTE  L4-L1     ;COEFFICIENT
0C              .BYTE  L5-L4
18              .BYTE  L6-L1     ;OUT OF ALLOWED RANGE
15              .BYTE  L7-L6
00              .BYTE  L1 - L1   ;REENTER
0A              .BYTE  L3-L1
FF              .BYTE  X'FF
35      I3:     .BYTE  L8-L1     ;ANSWER=
08              .BYTE  L10-L8
FF              .BYTE  X'FF
2D      I4:     .BYTE  L7-L1     ;ANOTHER?
08              .BYTE  L8-L7
FF              .BYTE  X'FF
3D      I5:     .BYTE  L10-L1    ;NO VALID ANSWER
09              .BYTE  L11-L10
35              .BYTE  L8-L1
06              .BYTE  L9-L8
00              .BYTE  L1-L1     ;.REENTER 5 COEFFICIENTS
18              .BYTE  L6-L1
FF              .BYTE  X'FF
```

A subroutine generates the printed messages. To write a message, the procedure is to call the subroutine and to specify the index that identifies the message to be printed. For example, to print "Answer=", the call would be as shown below.

```
JSR     WRIT      ;WRITE "ANSWER="
.DBYTE  I3        ;RETURN POINT
```

Subroutine WRIT calculates the section of the literal pool to be printed. The first character is at the address: L1 plus the contents of I3. The number of characters to be printed is derived from the contents of I3 + 1. The next byte contains X'FF, so printing is finished; otherwise, printing would continue with the next pair of index bytes specifying the next string of characters.

7.5 INPUT AND OUTPUT PROGRAMMING TECHNIQUES

The programming of data transfers between read/write memory and peripheral devices is generally classified as input/output programming. Depending on the significance of the input/output operations in the overall program, different approaches to input/output program implementation are recommended; these approaches are described in the following sections.

7.5.1 Programmed Input/Output

A programmed input/output operation is initiated and completed under the control of the initiating program. In figure 7-1, the program being executed starts the input/output operation; then, the program waits for the operation to be completed before continuing.

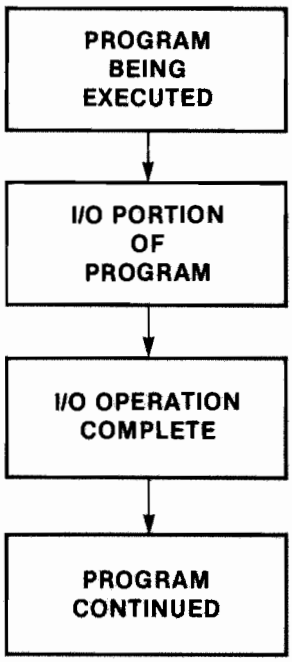


Figure 7-1. Programmed Input/Output

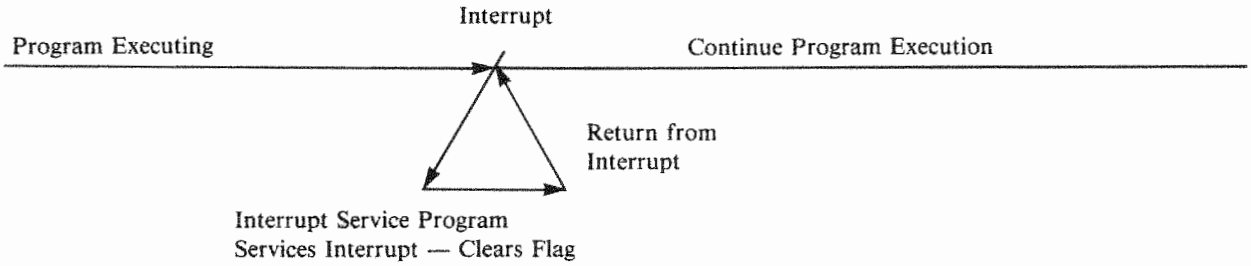
The 807X allows any memory-reference instruction to execute a programmed input/output operation. Peripheral device controllers are assigned specific memory addresses, any of which, when referenced by a memory-reference instruction, executes the input/output operation. It is necessary that the memory addresses assigned to the peripheral device be unique to the device; that is, no other peripheral device uses the same assigned memory addresses, nor is there memory with the same addresses. Also, the device controller must contain the necessary logic to decode its assigned addresses, and, then, to gate data on and off the data bus. For example, memory addresses X'7FFF and below might be reserved for read/write or read-only memory and memory addresses X'8000 to X'8FFF might be reserved for peripheral device controllers. The user, however, can define his own convention, as described in the 807X Data Sheet.

The actual program steps required to enable programmed input/output depend on the design of the device controller.

7.5.2 Interrupt Input/Output

In certain cases, an input/output operation initiated by a program requires a significant length of time (many milliseconds) for execution; during this time, the program might perform other tasks. In other cases, the frequency of input/output service that requires the use of a certain input/output device might be such that it would be convenient for the program to ignore the device unless it specifically requires service. Each of these situations may be handled by taking advantage of the 807X interrupt system and by employing interrupt input/output for devices that have interrupt capability.

In figure 7-2, the program might initiate the input/output operation as part of its normal sequence-of-operation and might set a flag indicating that such action was taken. An input/output device that has interrupt capability, upon completion of an input/output, transmits an interrupt to 807X to indicate completion of the operation.



The flag may be employed by the original program to determine whether or not the input/output operation has been completed and whether or not the input/output device is still busy.

Another way that an input/output operation may be initiated is to transmit an interrupt signal to the CPU. In this case, the program being executed is interrupted, the input/output operation is effected, and, then, the interrupted program is resumed.

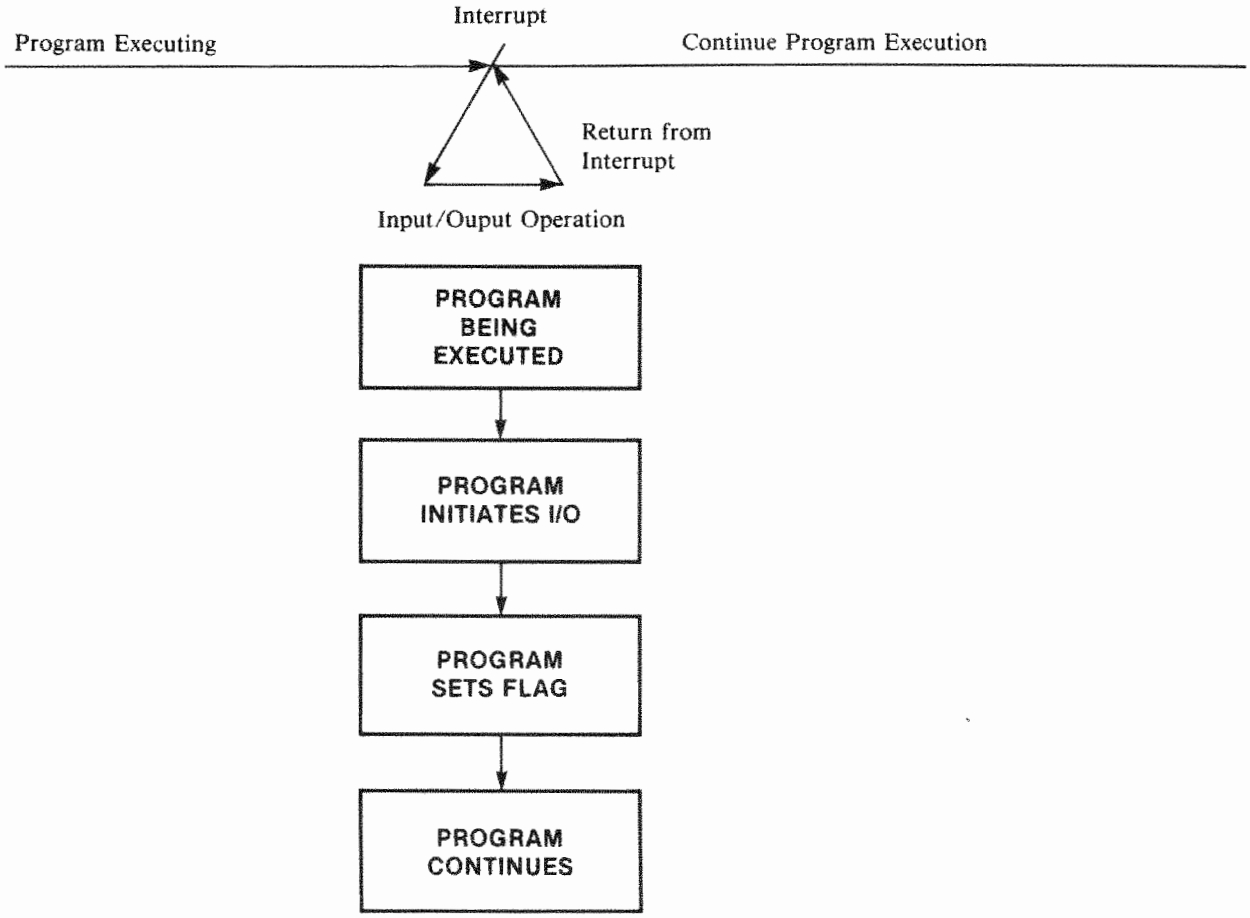


Figure 7-2. Interrupt Input/Output Initiation

Chapter 8

70-SERIES APPLICATIONS

At the time of writing, the following applications have not been fully investigated; however, they represent a cross section of typical systems where the 70-series family offers a highly efficient, low cost solution.

8.1 TERMINALS, PERSONAL COMPUTERS, AND VIEWDATA SETS

These three applications contain identical requirements of data display via a CRT. System cost is drastically reduced due to simplified buffering and synchronization requirements between the display controller system RAM and CPU. Typically, a dumb terminal may be implemented with a total of less than 18 IC's. The general configuration is shown in figure 8-1.

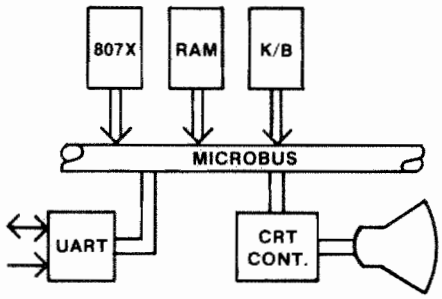


Figure 8-1. General Configuration for Terminals, Personal Computers, and Viewdata Sets

8.2 MULTIPROCESSING SYSTEMS

Through the use of the on-board executable RAM, and by taking advantage of the high-speed 16-bit arithmetic operations, real-time solutions of polynomials become extremely simple and economical.

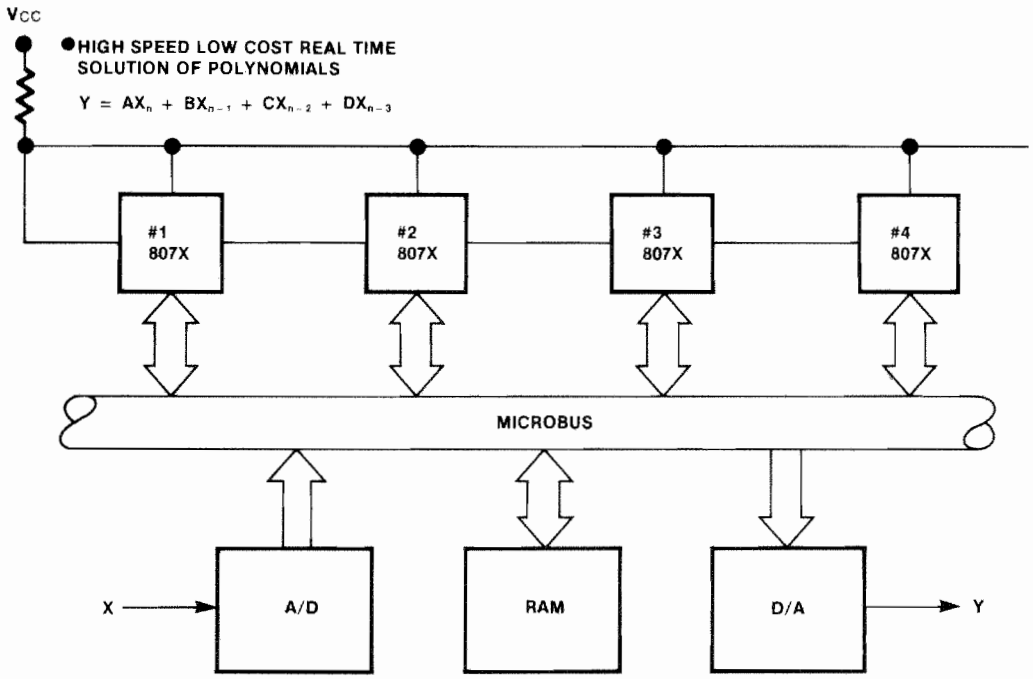


Figure 8 – 2. Multiprocessing Systems

8.3 INTELLIGENT PERIPHERAL CONTROLLER

One of the unconventional aspects of the INS807X family is the manner in which they can be used as add-on processors for such applications as intelligent I/O channels or real-time control processors.

The fact that they do not "listen" to NRDS and NWDS (MICROBUS $\overline{RD}$ and $\overline{WR}$) makes it seem as if command and status transfers would be cumbersome to implement. In fact, this would only be a disadvantage if the command, status, mode, etc. registers were *internal* to the chip. If these registers are instead allowed to reside in memory as "mailboxes", i.e., shared communication areas, this problem virtually disappears. The only questions remaining involve throughput efficiency and cost effectiveness.

Let's take an illustration, using MICROBUS terminology.

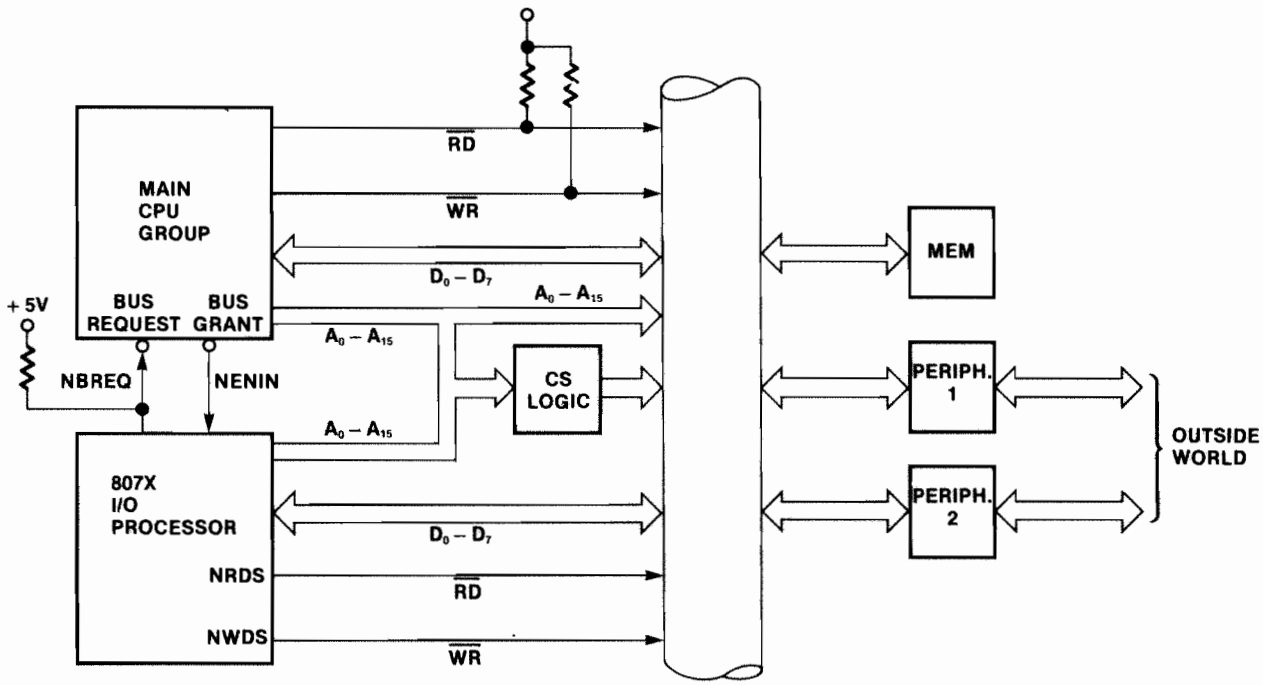


Figure 8 – 3. Typical 807X to System Connection

In this, the simplest configuration for an add-on multiprocessing system, both processors have full access to memory and peripherals. This configuration is especially desirable either where maintainability is critical (the I/O processor can easily be disabled by gating out NBREQ, allowing diagnostic procedures to be run directly through the main CPU Group), or in situations where the CPU's bus utilization percentage is low (e.g., a CPU Group consisting of an 8073 or 8075 HLL Processor, which has an expected bus utilization of around 2%). More hardware may be added to restrict or redefine the I/O Processor's addressing range, but in either case the following discussion holds.

Given a setup as described above, the burning question is: "How can the CPU and I/O Processor *efficiently* exchange command and status information?" The mailbox approach by itself will work, but if the two processors must constantly poll those locations in memory, the results are wasted time at both ends and increased bus utilization, both of which decrease potential throughput. The answer resides in the 8070 family's interrupt structure. The Sense A and Sense B inputs, when enabled as interrupts, allow a high-to-low transition to interrupt the internal program. This means that a simple I/O or Memory Read or Write may be used by the CPU to alert the I/O Processor that a command is waiting for it, with this little spent in overhead (see figure 8-4).

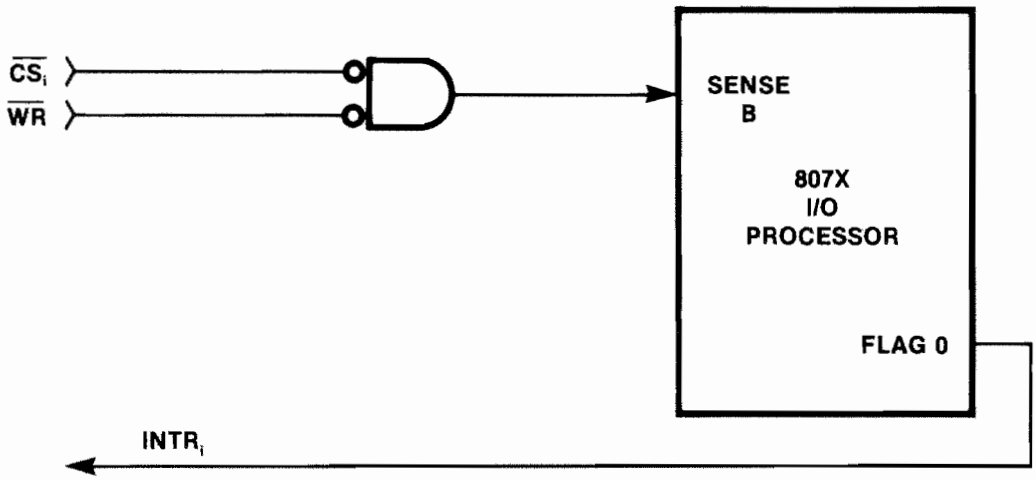


Figure 8 – 4. Interrupt and Acknowledge

The Chip Select input may be taken from an otherwise unused addressing space.

In a situation like this, all that the CPU has to do to send a command to the I/O Processor is (8070 mnemonics) =

```
LD      A,COMMAND      :Get command.
ST      A,IOCMD        ;Store it in mailbox.
ST      A,IOATTN       ;Alert I/O Processor.
```

and then refrain from sending further commands to the I/O Processor until the completion interrupt is received. More advanced programming techniques, if applied to the I/O Processor, can allow another responding interrupt to be generated as an acknowledgement of the receipt of a command, with the I/O Processor maintaining a queue of tasks to be performed. This can keep the CPU from having to wait as often, but it may require a careful worst-case analysis if high-speed peripherals are involved.

Another refinement of the above technique, using a bit more hardware, is to generate CSi not from an unused addressing space, but rather by detecting the command mailbox address. This eliminates the need for the extra alerting "Store" or "Output" instruction and makes programming the I/O Processor as straight-forward as if it were a memory-mapped conventional peripheral. Only one word of warning: the I/O Processor must not then write into its own command mailbox, since it would tend to interrupt itself.

Appendix A

70-Series Family Instruction Set

Appendix A contains the instruction set compatible to all the 70-Series devices described in this handbook. The instruction set is defined by function and delineates the mnemonic, operand, address and the operation performed.

INSTRUCTION	MNEMONIC	FIRST OPERAND	SECOND OPERAND	ADDRESSING: PC RELATIVE	POINTER RELATIVE: SP	POINTER RELATIVE: P2	POINTER RELATIVE: P3	IMMEDIATE	DIRECT	AUTO-INDEXED: P2	AUTO-INDEXED: P3	IMPLIED	INDEXED: P2	INDEXED: P3	ABSOLUTE or INDIRECT	FLAGS AFFECTED: CY	OV	SB	SA	F3	F2	F1	IE	OPERATION PERFORMED
				OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW	OP MC BY RW									
LOAD	LD	A		C0 7 2 3	C1 7 2 3	C2 7 2 3	C3 7 2 3	C4 5 2 2	C5 7 2 3	C6 9 2 3	C7 9 2 3													(A) ← (addr)
		EA		80 10 2 4	81 10 2 4	82 10 2 4	83 10 2 4	84 8 3 3	85 10 2 4	86 12 2 4	87 12 2 4													(EA) ← (addr 1 addr)
		T		A0 10 2 4	A1 10 2 4	A2 10 2 4	A3 10 2 4	A4 8 3 3	A5 10 2 4	A6 12 2 4	A7 12 2 4													(T) ← (addr 1 addr)
		SP						25 9 3 3																(SP) ← (PC 2, PC 1)
		P2						26 9 3 3																(P2) ← (PC 2, PC 1)
		P3						27 9 3 3																(P3) ← (PC 2, PC 1)
		A	E									40 4 1 1												(A) ← (E)
		E	A									48 4 1 1												(E) ← (A)
		A	S									06 3 1 1												(A) ← (S)
		S	A									07 3 1 1												(S) ← (A)
		EA	PC									30 4 1 1												(EA) ← (PC)
		EA	SP									31 4 1 1												(EA) ← (SP)
		EA	P2									32 4 1 1												(EA) ← (P2)
		EA	P3									33 4 1 1												(EA) ← (P3)
		SP	EA									45 5 1 1												(SP) ← (EA)
		P2	EA									46 5 1 1												(P2) ← (EA)
		P3	EA									47 5 1 1												(P3) ← (EA)
		T	EA									09 4 1 1												(T) ← (EA)
		EA	T									0B 4 1 1												(EA) ← (T)
STORE	ST	A		C8 7 2 3	C9 7 2 3	CA 7 2 3	CB 7 2 3		CD 7 2 3	CE 9 2 3	CF 9 2 3													(A) → (addr)
		EA		88 10 2 4	89 10 2 4	8A 10 2 4	8B 10 2 4		8D 10 2 4	8E 12 2 4	8F 12 2 4													(EA) → (addr 1 addr)
ADD	ADD	A		F0 7 2 3	F1 7 2 3	F2 7 2 3	F3 7 2 3	F4 5 2 2	F5 7 2 3	F6 9 2 3	F7 9 2 3					•	•							(A) ← (A) + (addr)
		A	E									70 4 1 1				•	•							(A) ← (A) + (E)
		EA		B0 10 2 4	B1 10 2 4	B2 10 2 4	B3 10 2 4	B4 8 2 3	B5 10 2 4	B6 12 2 4	B7 12 2 4					•	•							(EA) ← (EA) + (addr 1 addr)
SUBTRACT	SUB	A		F8 7 2 3	F9 7 2 3	FA 7 2 3	FB 7 2 3	FC 5 2 2	FD 7 2 3	FE 9 2 3	FF 9 2 3					•	•							(A) ← (A) - (addr)
		A	E									78 4 1 1				•	•							(A) ← (A) - (E)
		EA		B8 10 2 4	B9 10 2 4	BA 10 2 4	BB 10 2 4	BC 8 3 3	BD 10 2 4	BE 12 2 4	BF 12 2 4					•	•							(EA) ← (EA) - (addr 1 addr)
MULTIPLY	MPY	EA	T									2C 37 1 1				•	•							(EA) ← [(EA) * (T)] 31:16; (T) ← [(EA) * (T)] 15:0
DIVIDE	DIV	EA	T									0D 43 1 1				•	•							Quotient, (EA) ← [(EA) ÷ (T)] 15:0
AND	AND	A		D0 7 2 3	D1 7 2 3	D2 7 2 3	D3 7 2 3	D4 5 2 2	D5 7 2 3	D6 9 2 3	D7 9 2 3													(A) ← (A) ∧ (addr)
		A	E									50 4 1 1												(A) ← (A) ∧ (E)
		S						39 5 2 2								•	•			•	•	•	•	(S) ← (S) ∧ data
OR	OR	A		D8 7 2 3	D9 7 2 3	DA 7 2 3	DB 7 2 3	DC 7 2 2	DD 7 2 3	DE 9 2 3	DF 9 2 3													(A) ← (A) ∨ (addr)
		A	E									58 4 1 1												(A) ← (A) ∨ (E)
		S						3B 5 2 2								•	•			•	•	•	•	(S) ← (S) ∨ data
EXCLUSIVE-OR	XOR	A		E0 7 2 3	E1 7 2 3	E2 7 2 3	E3 7 2 3	E4 5 2 2	E5 7 2 3	E6 9 2 3	E7 9 2 3													(A) ← (A) ⊕ (addr)
		A	E									60 4 1 1												(A) ← (A) ⊕ (E)

70-Series Family Instruction Set - Continued

INSTRUCTION	MNEMONIC	FIRST OPERAND	SECOND OPERAND	ADDRESSING: PC RELATIVE (OP MC BY RW)	POINTER RELATIVE: SP (OP MC BY RW)	POINTER RELATIVE: P2 (OP MC BY RW)	POINTER RELATIVE: P3 (OP MC BY RW)	IMMEDIATE (OP MC BY RW)	DIRECT (OP MC BY RW)	AUTO-INDEXED: P2 (OP MC BY RW)	AUTO-INDEXED: P3 (OP MC BY RW)	IMPLIED (OP MC BY RW)	INDEXED: P2 (OP MC BY RW)	INDEXED: P3 (OP MC BY RW)	ABSOLUTE or INDIRECT (OP MC BY RW)	FLAGS AFFECTED: CY	OV	SB	SA	F3	F2	F1	IE	OPERATION PERFORMED
EXCHANGE REGISTERS	XCH	A	E									01 5 1 1												(A) ↔ (E)
		EA	SP									4D 7 1 1												(EA) ↔ (SP)
		EA	P2									4E 7 1 1												(EA) ↔ (P2)
		EA	P3									4F 7 1 1												(EA) ↔ (P3)
SHIFT RIGHT	SR	A										3C 3 1 1												(Ai) ← (Ai-1), i = 1, 7, 0 → A7
		EA										0C 4 1 1												(EAi) ← (EAi-1), i = 1, 15, 0 → EA15
SHIFT RIGHT WITH LINK	SRL	A										3D 3 1 1				•								(Ai) ← (Ai-1), i = 1, 7, CY L → A7
ROTATE RIGHT	RR	A										3E 3 1 1												(Ai) ← (Ai-1), i = 1, 7, A0 → A7
ROTATE RIGHT WITH LINK	RRL	A										3F 3 1 1				•								(Ai) ← (Ai-1), i = 1, 7, A0 → CY L → A7
SHIFT LEFT	SL	A										0E 3 1 1												(Ai+1) ← (Ai), i = 6, 0, 0 → A0
		EA										0F 4 1 1												(EAi+1) ← (EAi), i = 14, 0, 0 → EA0
SEARCH AND SKIP IF CHARACTER MATCHED	SSM												2E 1	2F 1										See Text
BRANCH IF NOT DIGIT	BND			2D 2																				See Text
PUSH	PUSH	A										0A 5 1 2												(SP) ← (SP) +1, ((SP)) ← (A)
		EA										08 8 1 3												((SP) +1) ← (E), ((SP) +2) ← (A), (SP) ← (SP) +2
		PC										54 8 1 3												((SP) +1) ← (PCH), ((SP) +2) ← (PCL), (SP) ← (SP) +2
		P2										56 8 1 3												((SP) +1) ← (P2H), ((SP) +2) ← (P2L), (SP) ← (SP) +2
		P3										57 8 1 3												((SP) +1) ← (P3H), ((SP) +2) ← (P3L), (SP) ← (SP) +2
PUSH AND LOAD IMMEDIATE	PLI	P2						22 16 3 5																((SP) +1) ← (P2H); ((SP) +2) ← (P2L); (SP) ← (SP) +2 (P2H) ← byte 3, (P2L) ← byte 2
		P3						23 16 3 5																((SP) +1) ← (P3H), ((SP) +2) ← (P3L), (SP) ← (SP) +2 (P3H) ← byte 3; (P3L) ← byte 2
POP	POP	A										38 6 1 2												(A) ← ((SP)); ← (SP) +1
		EA										3A 9 1 3												(A) ← ((SP)), (E) ← ((SP) +1); (SP) ← (SP) +2
		P2										5E 10 1 3												(P2L) ← ((SP)); (P2H) ← ((SP) +1); (SP) ← (SP) +2
		P3										5F 10 1 3												(P3L) ← ((SP)); (P3H) ← ((SP) +1); (SP) ← (SP) +2
BRANCH UNCONDITIONAL	BRA			74 5 2 2		76 5 2 2	77 5 2 2																	(PC) ← (Pn) + disp
BRANCH POSITIVE	BP			64 5 2 2		66 5 2 2	67 5 2 2																	If (A) > 0: (PC) ← (Pn) + disp
BRANCH ZERO	BZ			6C 5 2 2		6E 5 2 2	6F 5 2 2																	If (A) = 0, (PC) ← (Pn) + disp
BRANCH NOT ZERO	BNZ			7C 5 2 2		7E 5 2 2	7F 5 2 2																	If (A) ≠ 0: (PC) ← (Pn) + disp
JUMP UNCONDITIONAL	JMP														24 9 3 3									(PCH) ← byte 3; (PCL) ← byte 2
JUMP TO SUBROUTINE	JSR														20 16 3 3									((SP) +1) ← (PCH) ← byte 3, ((SP) +2) ← (PCL) ← byte 2, (SP) ← (SP) +2
CALL	CALL	0 - 15													10-1F 16 1 5									((SP) +1) ← (PCH) ← (021 + 2*N); ((SP) +2) ← (PCL) ← (020 + 2*N); (SP) ← (SP) +2
RETURN	RET											5C 10 1 3												(PCL) ← ((SP)); (PCH) ← ((SP) +1); (SP) ← (SP) +2
LOAD PC	LD	PC	EA									44 5 1 1												(PC) ← (EA)
EXCHANGE PC	XCH	PC	EA									4C 7 1 1												(PC) ↔ (EA)
INCREMENT AND LOAD	ILD	A		90 8 2 4	91 8 2 4	92 8 2 4	93 8 2 4		95 8 2 4	96 10 2 4	97 10 2 4													(A), (addr) ← (addr) +1
DECREMENT AND LOAD	DLD	A		98 8 2 4	99 8 2 4	9A 8 2 4	9B 8 2 4		9D 8 2 4	9E 10 2 4	9F 10 2 4													(A), (addr) ← (addr) -1
NO OPERATION	NOP											00 3 1 1												(PC) ← (PC) +1

Appendix B
70-Series Family Op Code Listing

HEX	MNEMONIC		HEX	MNEMONIC		HEX	MNEMONIC		HEX	MNEMONIC	
00	NOP		20	JSR		80	LD	EA,,PC	A0	LD	T,,PC
01	XCH	A, E	21	—		81	LD	EA,,SP	A1	LD	T,,SP
02	—		22	PLI	P2, = ADDR	82	LD	EA,,P2	A2	LD	T,,P2
03	—		23	PLI	P3, = ADDR	83	LD	EA,,P3	A3	LD	T,,P3
04	—		24	JMP		84	LD	EA, = DATA 2	A4	LD	T, = DATA 2
05	—		25	LD	SP, = ADDR	85	LD	EA,DIR ADDR	A5	LD	T,DIR ADDR
06	LD	A, S	26	LD	P2, = ADDR	86	LD	EA,,@P2	A6	LD	T,,@P2
07	LD	S, A	27	LD	P3, = ADDR	87	LD	EA,,@P3	A7	LD	T,,@P3
08	PUSH	EA	28	—		88	ST	EA,,PC	A8	—	
09	LD	T, EA	29	—		89	ST	EA,,SP	A9	—	
0A	PUSH	A	2A	—		8A	ST	EA,,P2	AA	—	
0B	LD	EA, T	2B	—		8B	ST	EA,,P3	AB	—	
0C	SR	EA	2C	MPY	EA, T	8C	—		AC	—	
0D	DIV	EA, T	2D	RND		8D	ST	EA,DIR ADDR	AD	—	
0E	SL	A	2E	SSM	P2	8E	ST	EA,,@P2	AE	—	
0F	SL	EA	2F	SSM	P3	8F	ST	EA,,@P3	AF	—	
10	CALL	0	30	LD	EA, PC	90	ILD	A,,PC	B0	ADD	EA,,PC
11	CALL	1	31	LD	EA, SP	91	ILD	A,,SP	B1	ADD	EA,,SP
12	CALL	2	32	LD	EA, P2	92	ILD	A,,P2	B2	ADD	EA,,P2
13	CALL	3	33	LD	EA, P3	93	ILD	A,,P3	B3	ADD	EA,,P3
14	CALL	4	34	—		94	—		B4	ADD	EA, = DATA 2
15	CALL	5	35	—		95	ILD	A,DIR ADDR	B5	ADD	EA,DIR
16	CALL	6	36	—		96	ILD	A,,@P2	B6	ADD	EA,,@P2
17	CALL	7	37	—		97	ILD	A,,@P3	B7	ADD	EA,,@P3
18	CALL	8	38	POP	A	98	DLD	A,,PC	B8	SUB	EA,,PC
19	CALL	9	39	AND	S, = DATA 1	99	DLD	A,,SP	B9	SUB	EA,,SP
1A	CALL	10	3A	POP	EA	9A	DLD	A,,P2	BA	SUB	EA,,P2
1B	CALL	11	3B	OR	S, = DATA 1	9B	DLD	A,,P3	BB	SUB	EA,,P3
1C	CALL	12	3C	SR	A	9C	—		BC	SUB	EA, = DATA 2
1D	CALL	13	3D	SRL	A	9D	DLD	A,DIR ADDR	BD	SUB	EA,DIR ADDR
1E	CALL	14	3E	RR	A	9E	DLD	A,,@P2	BE	SUB	EA,@P2
1F	CALL	15	3F	RRL	A	9F	DLD	A,,@P3	BF	SUB	EA,@P3
40	LD	A, E	60	XOR	A, E	C0	LD	A,, = DISPL PC	E0	XOR	A,,PC
41	—		61	—		C1	LD	A,, = DISPL SP	E1	XOR	A,,SP
42	—		62	—		C2	LD	A,, = DISPL P2	E2	XOR	A,,P2
43	—		63	—		C3	LD	A,, = DISPL P3	E3	XOR	A,,P3
44	LD	PC, EA	64	BP		C4	LD	A, = DATA 1	E4	XOR	A, = DATA 1
45	LD	SP, EA	65	—		C5	LD	A,DIR ADDR	E5	XOR	A,DIR ADDR
46	LD	P2, EA	66	BP	P2	C6	LD	A,,@P2	E6	XOR	A,,@P2
47	LD	P3, EA	67	BP	P3	C7	LD	A,,@P3	E7	XOR	A,,@P3
48	LD	E, A	68	—		C8	ST	A,,PC	E8	—	
49	—		69	—		C9	ST	A,,SP	E9	—	
4A	—		6A	—		CA	ST	A,,P2	EA	—	
4B	—		6B	—		CB	ST	A,,P3	EB	—	
4C	XCH	EA, PC	6C	BZ		CC	—		EC	—	
4D	XCH	EA, SP	6D	—		CD	ST	A,DIR ADDR	ED	—	
4E	XCH	EA, P2	6E	BZ	P2	CE	ST	A,,@P2	EE	—	
4F	XCH	EA, P3	6F	BZ	P3	CF	ST	A,,@P3	EF	—	
50	AND	A, E	70	ADD	A, E	D0	AND	A,,PC	F0	ADD	A,,PC
51	—		71	—		D1	AND	A,,SP	F1	ADD	A,,SP
52	—		72	—		D2	AND	A,,P2	F2	ADD	A,,P2
53	—		73	—		D3	AND	A,,P3	F3	ADD	A,,P3
54	PUSH	PC	74	BRA		D4	AND	A, = DATA 1	F4	AND	A, = DATA 1
55	—		75	—		D5	AND	A,DIR ADDR	F5	ADD	A,DIR ADDR
56	PUSH	P2	76	BRA	P2	D6	AND	A,,@P2	F6	ADD	A,,@P2
57	PUSH	P3	77	BRA	P3	D7	AND	A,,@P3	F7	ADD	A,,@P3
58	OR	A, E	78	SUB	A, E	D8	OR	A,,PC	F8	SUB	A,,PC
59	—		79	—		D9	OR	A,,SP	F9	SUB	A,,SP
5A	—		7A	—		DA	OR	A,,P2	FA	SUB	A,,P2
5B	—		7B	—		DB	OR	A,,P3	FB	SUB	A,,P3
5C	RET		7C	BNZ		DC	OR	D, = DATA 1	FC	SUB	A, = DATA 1
5D	—		7D	—		DD	OR	A, DIR ADDR	FD	SUB	A,DIR ADDR
5E	POP	P2	7E	BNZ	P2	DE	OR	A,,@P2	FE	SUB	A,,@P2
5F	POP	P3	7F	BNZ	P3	DF	OR	A,,@P3	FF	SUB	A,,@P3

NOTES:
1. ADDR = 2 bytes of address
2. DATA 1 = 1 byte of data
3. DATA 2 = 2 bytes of data
4. DISPL = 1 byte (-128_D to $+127_D$)
5. DIR ADDR = 1 byte that represents lower byte of direct address Upper byte of direct address is fixed at address X'FF

Appendix C

INDEX OF DIRECTIVES

Table C-1. Alphabetic Index of Directives

Directives	Mnemonic	Operands
ASCII	.ASCII	string [,string...]
Byte	.BYTE	expression [,expression...]
Double byte	.DBYTE	expression [,expression...]
Do (1)	.DO	count
Conditional assembly	.ELSE	
End	.END	[address]
End Do-loop (1)	.ENDDO	
Conditional assembly	.ENDIF	
End Macro (1)	.ENDM	
Error	.ERROR	string
Exit Do (1)	.EXIT	
Conditional assembly	.IF	expression
Conditional assembly (2)	.IFC	string1 operator string2
Include	.INCLD	filename
List	.LIST	immediate
Local	.LOCAL	
Macro (1)	.MACRO	mname [parameters]
Delete macro	.MDEL	mname [,mname...]
Macro local (1)	.MLOC	symbol [,symbol...]
New page	.PAGE	[string]
Set	.SET	symbol,expression
Space forward	.SPACE	immediate
Symbol table value	.STVAL	symbol1, symbol2
Title	.TITLE	symbol [,string]

NOTES: (1) Cannot be used outside of a macro definition.(2) In the .IFC directive, operator means a relational operator and may be either "EQ" for equal or "NE" for not equal.

Appendix D

Appendix D contains the INS8070-Series Family Microprocessor data sheets.

National Semiconductor

Preliminary
August 1979

INS8070-Series Microprocessor Family

General Description

The INS8070-Series of microprocessors (hereinafter referred to as the 70-Series Family) is intended for use in systems requiring the economy of a single chip, the flexibility of multiprocessing bus architecture and the power of 16-bit arithmetic operations.

Designed for simplicity and low-cost implementation for stand-alone multiprocessing and DMA systems, the members of the 70-Series family include:

INS8070 - 64 Bytes of RAM, no ROM
INS8072 - 64 Bytes of RAM, 2.5K ROM

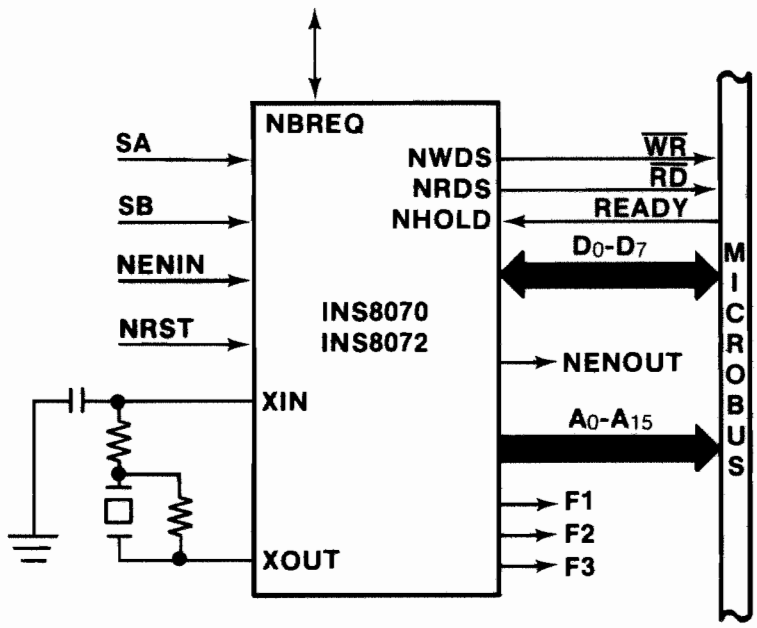
The 70-Series family comprises bus-oriented microprocessors with on-board ROM and RAM and built-in multi-processing logic. Low-cost systems containing data may occupy the same bus and still run at full speed, as their external bus requirements are confined to the occasional access of data. Such systems include terminals, intelligent peripherals and multiprocessing systems (e. g., the solution of polynomials in real-time).

The 70-Series family is fabricated using National's Standard N-channel, silicon gate, depletion load, MOS technology. All the devices are housed in standard 40-pin dual-in-line packages.

Features

- On-Board ROM
- On-Board RAM (Executable)
- 8-Bit Data Bus
- 16-Bit Address Bus (64K Addressing Capability)
- Comprehensive Set of 8 and 16-Bit Arithmetic, Logic and Stack Manipulation Instructions
- Hardware 16 x 16 Bit Multiply (37 μs) and Divide (42 μs)
- Built-in Multiprocessing and DMA Logic
- Interfaces with Memory and Standard INS8080 Peripherals at Any Clock Speed
- On-Chip Clock Generation
- Single Instruction Character Search and Single Instruction ASCII to Decimal Conversion
- Full Hardware and Software Development Systems Will be Available
- Single 5-Volt Supply
- MICROBUS™ Compatible

70-Series Family to MICROBUS Configuration



© 1979 National Semiconductor Corp.

Absolute Maximum Ratings

Max. Voltage to Any Input with Respect to GND.... -0.5V to +7.0V
Operating Temperature ..... 0° C to +70° C
Storage Temperature .....-65° C to 70° C

NOTE: *Maximum ratings indicate limits beyond which permanent damage may occur. Continuous operation at these limits is not intended and should be limited to those conditions specified under DC electrical characteristics.*

DC Electrical Characterisitics

(T_A = 0° C to +70° C, V_{CC} = +5V ±5%)

Parameter	Min.	Typ	Max	Units	Conditions
INPUT SPECIFICATIONS					
All Input Pins Except V_{CC} and GND, NRST, and Sense Inputs SA and SB (See Note 1.) Logic 1 Input Voltage	2.0		V_{CC}	V	
Logic 0 Input Voltage	-0.5		0.8	V	
Inputs with Schmitt Trigger Input Stage (NRST, SA, SB) V_T + Positive Going Threshold Voltage			2.2	V	
V_T - Negative Going Threshold Voltage	0.8			V	
Hysteresis (V_T+ - V_T-)	0.2			V	
Input Capacitance (All pins except V_{CC} and GND)		6		pF	
Supply Current I_{CC}		100 mA		mA	T_A = 25° C Outputs unloaded

NOTE 1: *SA and SB are not buffered by the Schmitt Trigger in the interrupt path and meet the normal Input Voltage specifications when used as Interrupt Inputs.*

OUTPUT SPECIFICATIONS

TRI-STATE® Pins (NWDS, NRDS, DB7-DB0, AD15-AD00) Logic 1 Output Voltage	2.4			V	I_{OUT} = -100 μA
Logic 0 Output Voltage			0.4	V	I_{OUT} = 1.6 mA
TRI-STATE Output Current			±10	μA	0 < V_{OUT} < V_{CC}
FLAG 1-3, NENOUT Logic 1 Output Voltage	V_{CC} -1			V	I_{OUT} = -100 μA
Logic 1 Output Voltage	1.5			V	I_{OUT} = -1 mA
Logic 0 Output Voltage			0.4	V	I_{OUT} = 1.6 mA
NBREQ Logic 0 Output Voltage			0.4	V	I_{OUT} = 1.6 mA
Logic 1 Output Current			±10	μA	0<V_{OUT}<V_{CC}
XOUT Logic 1 Output Voltage	2.4			V	I_{OUT} = -100 μA
Logic 0 Output Voltage			0.4	V	I_{OUT} = 1.6 mA

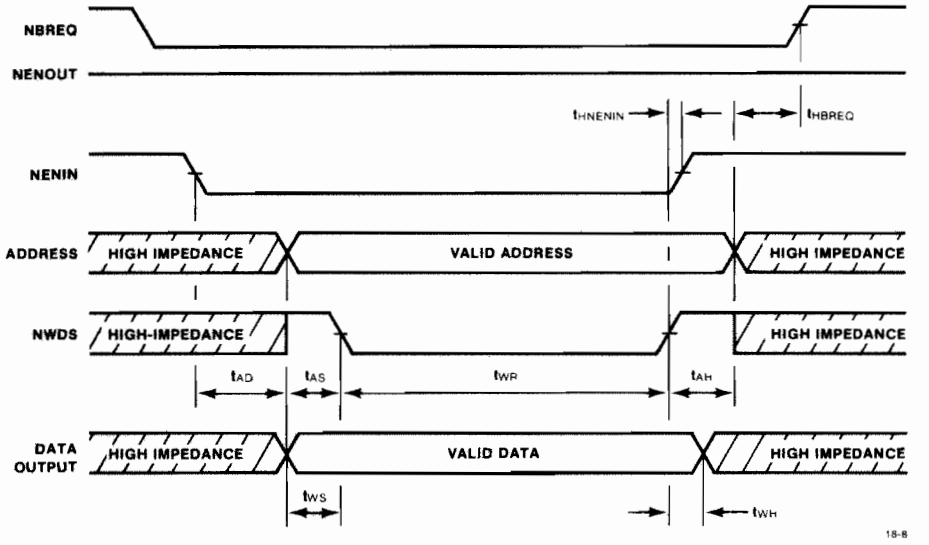
AC Electrical Characteristics

(T_A = +25°C, V_{CC} = +5V ±5%, GND = OV)

Parameter	Min.	Typ	Max.	Unit	Test Conditions (See Note)
EXTERNAL MEMORY f_x (Input Frequency)	0.1		4.0	MHz	Note: tx is the period (1) fx of the XIN input. When f_x = 4MHz, tx = 250ns.
Read Cycle					
t_{RD}		4tx		ns	
t_{AS}		tx		ns	
t_{AH}		tx		ns	
t_{ACC}			600	ns	
t_{DH}	0			ns	
t_{AD}		2tx		ns	
t_{HBREQ}		50		ns	
t_{HNENIN}	0			ns	
Write Cycle					
t_{WR}		3tx		ns	
t_{AS}		2tx		ns	
t_{AH}		tx		ns	
t_{WS}		tx		ns	
t_{WH}		tx		ns	
t_{AD}		2tx		ns	
t_{HBREQ}		50		ns	
t_{HNENIN}	0			ns	
EXTERNAL READ/WRITE CYCLE EXTENSION					See *Figure 3*
t_{D1} (HOLD) Read Cycle			2tx	ns	
Write Cycle			tx	ns	
t_W (HOLD)			∞	ns	
t_S (HOLD)	2tx			ns	
t_{D2} (HOLD)		3tx		ns	
t_{D3} (HOLD)		4tx		ns	
Capacitances					
Output Load (Xout)		30		pF	
All Other Output Pins		75		pF	

Timing Waveforms

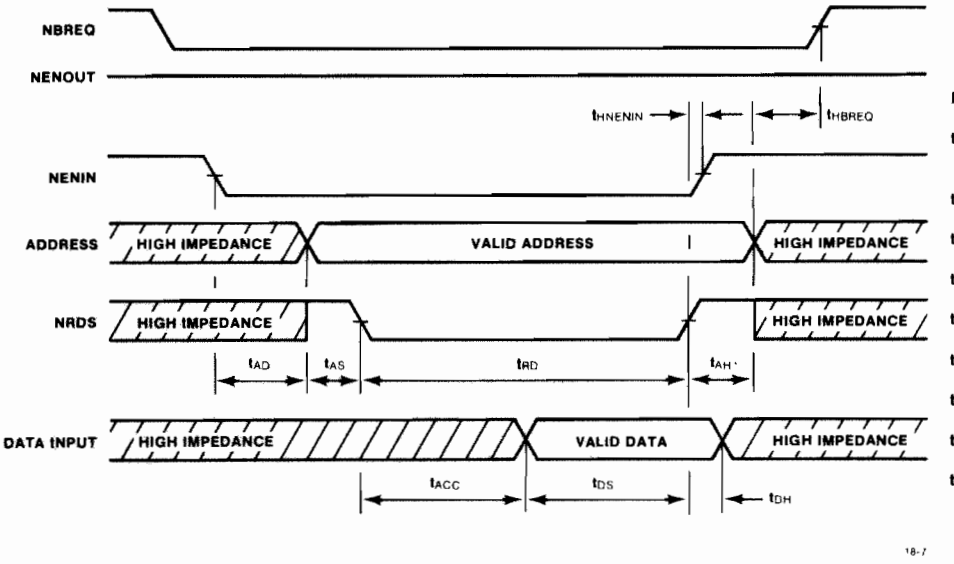
Write Cycle



WRITE CYCLE (fx = 4MHz max)

t_{WR}	≈	3tx ns typ
t_{AS}	≈	2tx ns typ
t_{AH}	≈	tx ns typ
t_{WS}	≈	tx ns typ
t_{WH}	≈	tx ns typ
t_{AD}	≈	2tx ns typ
t_{HBREQ}	≈	50 typ
t_{HNENIN}	≈	0

Read Cycle



READ CYCLE (fx = 4MHz max)

tx	=	$\frac{1}{fx}$
t_{RD}	≈	4tx ns typ
t_{AS}	≈	tx ns typ
t_{AH}	≈	tx ns typ
t_{ACC}	≈	600
t_{DH}	≈	0
t_{AD}	≈	2tx ns typ
t_{HBREQ}	≈	50 typ
t_{HNENIN}	≈	0

INS807X Functional Pin Description

The following describes the function of all INS807X input/output pins. For brevity, the INS807X designator is used to refer to all members of the family.

INPUT SIGNALS

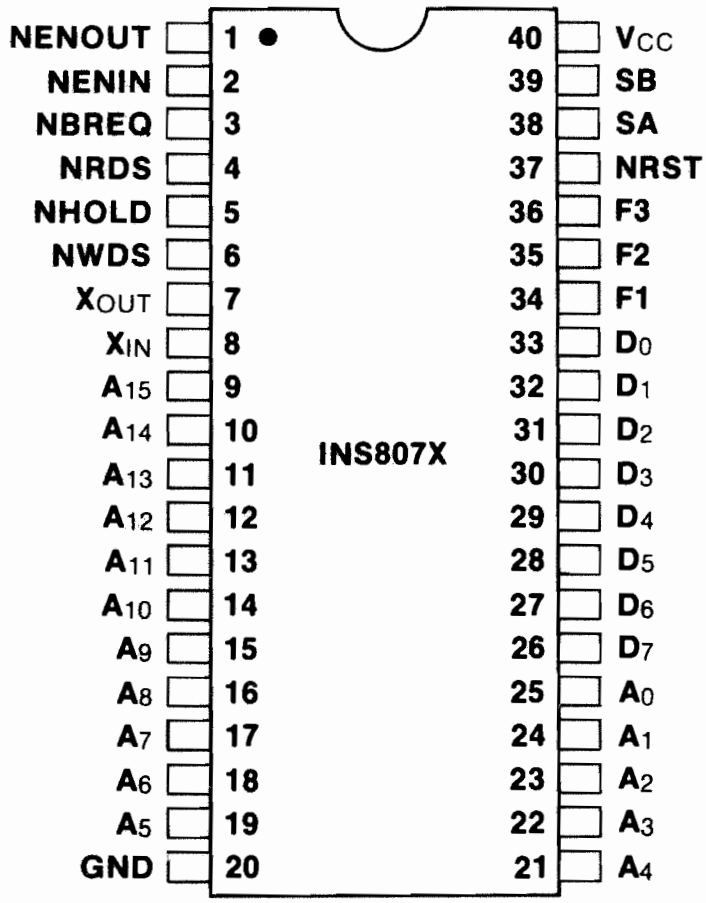
Reset	(NRST)	Active low RESET input. This pin initializes the INS807X by resetting the program counter (PC) and the stack pointer (SP) to zero (low). All bits in the Status Register (except SA and SB) are also reset to zero. This input is buffered by a T^2L compatible Schmitt trigger, allowing slow rise and fall times.
Hold	(NHOLD)	Active low external memory cycle extend input for slow memories and slow peripherals. Also used for signal memory cycle execution. NHOLD affects READ/WRITE cycles to external memory only.
Sense A/Interrupt A	(SA/INTA)	Provides a T^2L compatible Schmitt trigger buffered sense input which sets/resets the Sense A bit (SA) in the status register. When interrupts are enabled (IE = 1 in the STATUS register) this pin also acts as a trailing edge triggered interrupt input.
Sense B/Interrupt B	(SB/INTB)	Similar operation as SA/INTA. Since SA/INTA has priority over SB/INTB, if both arrive simultaneously, Interrupt A will be serviced first.
Enable Input	(NENIN)	Active low bus enable input to the on-chip bus allocation logic. For DMA and multiprocessing applications this logic allows bus access to be granted or denied to the INS807X. Devices which can share the bus include other INS807X processors, INS8060 processors, DP8350 CRT Controller and DMA logic. NENIN affects INS807X operation as follows: 1) If NENIN is high, INS807X sets NENOUT high and is denied access to the bus. 2) If NENIN is low and the INS807X is holding NBREQ low, INS807X sets NENOUT high and is granted access to the bus. 3) If NENIN and NBREQ are both low and the INS807X is not holding NBREQ low, INS807X sets NENOUT low, and is denied access to the bus.
Power		V_{CC} = +5V, Pin 40 V_{SS} = GND, Pin 20

OUTPUT SIGNALS

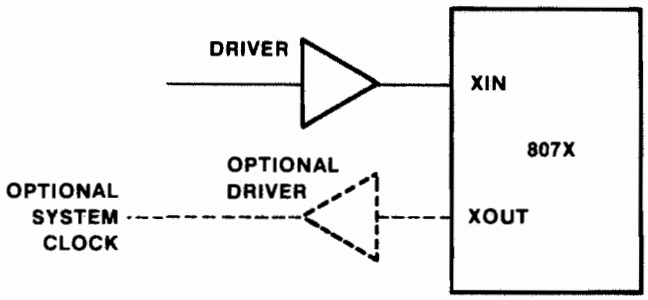
Address Bus	(A_{15} - A_0)	TRI-STATE address outputs for all external memory READ and WRITE operations. These outputs are normally in high-impedance state except during external memory READ and WRITE operations (Refer to Timing Waveforms.)
Write Data Strobe	(NWDS)	TRI-STATE active low WRITE strobe output. NWDS goes low during an external memory WRITE operation. NWDS is in the high-impedance state when an external memory WRITE operation is not in progress. (Refer to Write Cycle Timing.)
Read Data Strobe	(NRDS)	TRI-STATE active low READ strobe output. NRDS goes low during an external memory READ operation. NRDS is in the high-impedance state when an external memory READ operation is not in progress. (Refer to Read Cycle Timing.)
Enable Output	(NENOUT)	Active low bus enable output from the on-chip bus allocation logic. INS807X controls NENOUT as follows: 1) If NBREQ is low but INS807X is not holding it low, NENOUT = NENIN.

		2) If NEBREQ is low and INS807X is holding it low, NENOUT = high. 3) If NBREQ = high, NENOUT = high.
Flags 1, 2, 3	(F1, F2, F3)	Flag outputs which can be set high/low by writing into the corresponding flag bits of the status register.
	BIDIRECTIONAL SIGNALS	
Bus Request	(NBREQ)	NBREQ is a bidirectional line that serves as a bus status/request for the INS807X and any other devices sharing the data bus. The NBREQ pin must be externally connected to V_{CC} through a pullup resistor. When NBREQ acts as an input to the INS807X, a logic high indicates that the data bus (D0-D7) is available; a logic low signifies the data bus is unavailable. When NBREQ acts as an output from the INS807X, a low indicates that the INS807X is requesting access to the data bus; a high indicates that the data bus is available to other devices
Data Bus	(D_0 - D_7)	Bidirectional TRI-STATE data bus input/output. These lines are normally in the high-impedance state except during external memory READ and WRITE operations. They provide input data during external memory READ; they provide output data during external memory WRITE.
Clock Inputs	(X_{IN} X_{OUT})	On-chip clock generator input/outputs (see *Figure 1*). The on-chip clock generator will operate with external crystal or RC inputs connected between X_{IN} and X_{OUT}. The on-chip clock generator may be disabled by supplying a T^2L level clock input on X_{IN}. In all modes a buffered T^2L level clock output is always available at X_{OUT}.

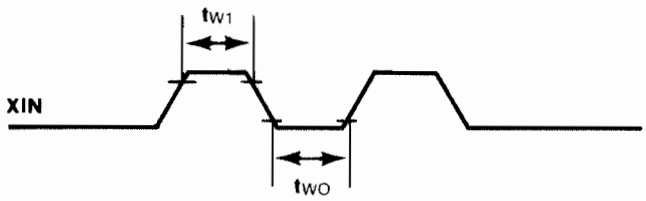
Pin Configuration



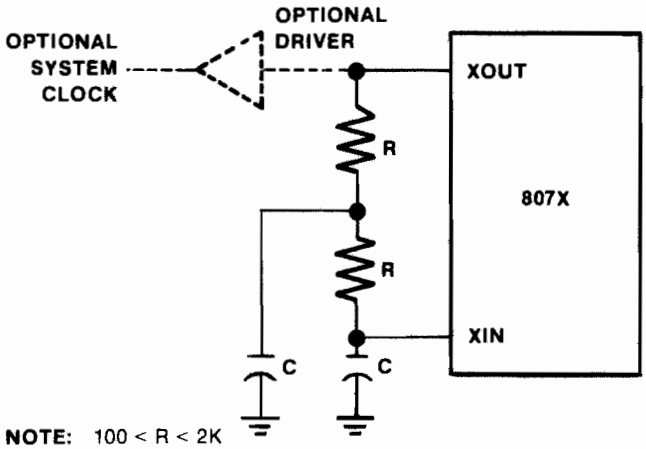
A. External Clock Input



EXTERNAL CLOCK PARAMETERS



B. Register Capacitor Feedback Network



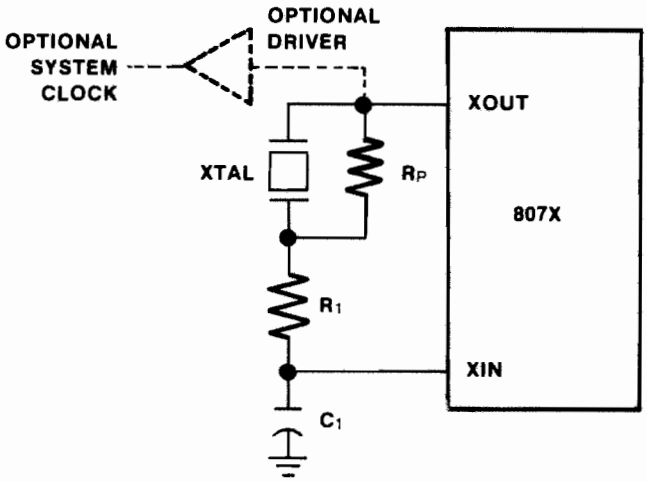
NOTE: 100 < R < 2K

Approximate Oscillator Frequency vs RC Time Constant

$$t = \frac{1}{f} = 0.725\ RC + 353$$

where: R is in KΩ
C is in pf
t is in ns

C. Crystal with Low Pass Filter (Above 1 MHz)

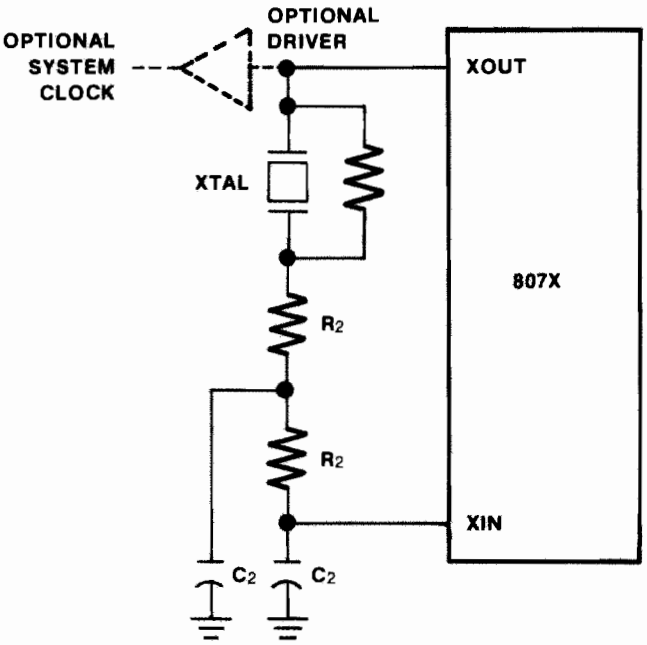


Typical values for Crystal with Low Pass Filter Network

Crystal	Rp	C_1	R_1
2 MHz	100kΩ	56pF	1kΩ
3.58 MHz	100kΩ	27pF	1kΩ
4 MHz	100kΩ	27pF	1kΩ

XTAL is parallel resonant with maximum series resonance equal to 1kΩ

D. Crystal with Low Pass Filter (1 MHz or Below)



NOTE: The values of R & C are determined by the operating frequency and crystal type.

FIGURE 1. Frequency Control Networks for On-Chip Oscillator

18-4

FUNCTIONAL DESCRIPTION

The INS8072 and INS8070 are members of a family of single chip, general-purpose microprocessors. The INS8072 contains 2.5K bytes of on-chip ROM and 64 bytes of on-chip RAM. The INS8070 contains no on-chip ROM and 64 bytes of on-chip RAM. The INS8072 and INS8070 may both be used in stand-alone, Direct Memory Access (DMA), and multiprocessor applications.

Communication between the 70-Series family and external memory/peripheral devices is effected via a 16-byte address bus and an 8-bit bidirectional data bus.

READ and WRITE strobe outputs from INS807X - (NRDS, NWDS) indicate when valid input/output data are present on the 8-bit data bus.

The remaining input/output signals shown in the block diagram are dedicated to initialization, clock generation, bus management, interrupt request, input/output cycle extension, and user-definable interface functions.

Drivers and Receivers

Equivalent circuits for INS807X drivers and receivers are shown in *Figure 2*. All inputs have static charge protection circuits consisting of an RC filter and voltage clamp. These devices should still be handled with care, as the protection circuits can be destroyed by excessive static charge.

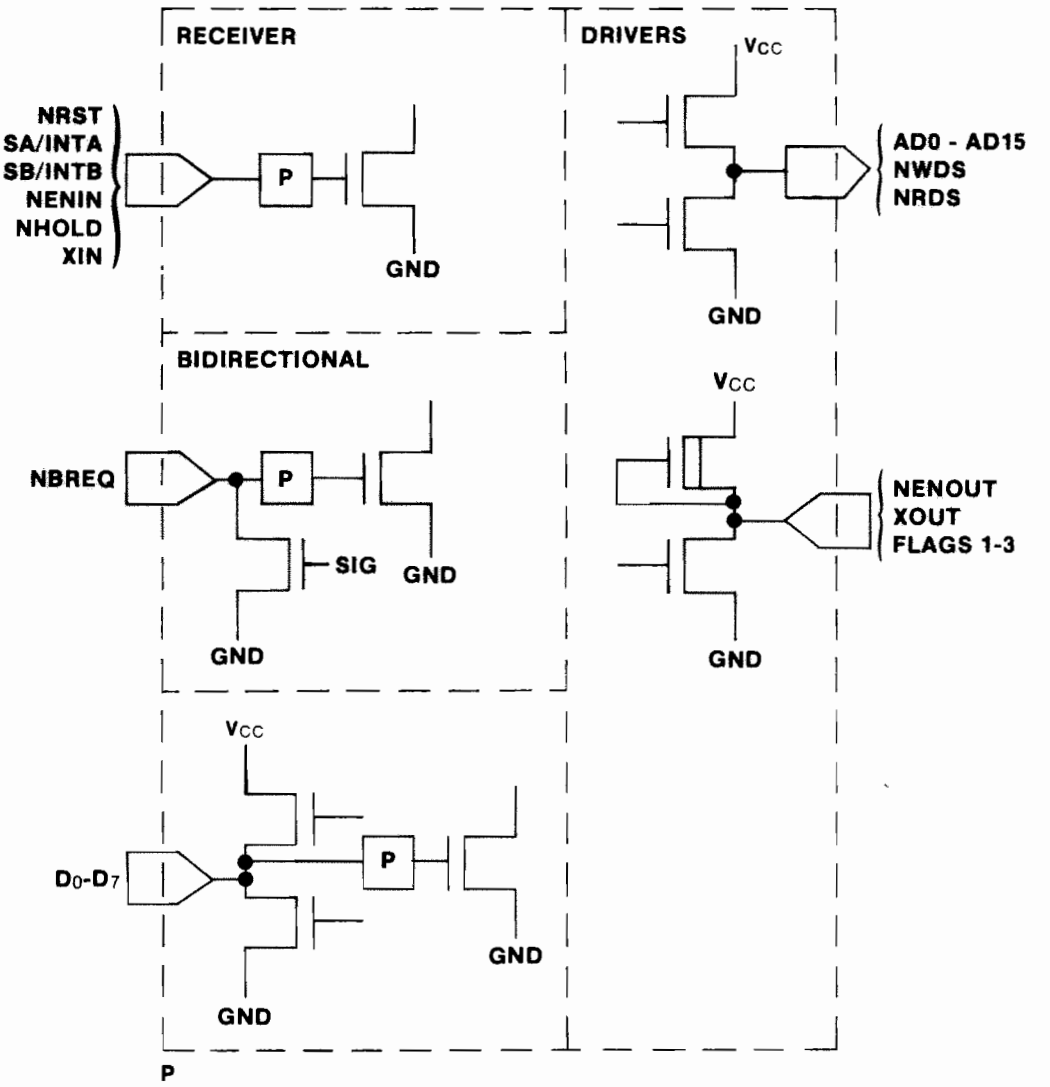


FIGURE 2. INS807X Driver and Receiver Equivalent Circuits

18-5

I/O Control Logic

The I/O Control Logic generates the basic timing for internal/external memory READ/WRITE operations. This includes extension of external READ/WRITE cycles in order to accomodate slow memory/peripherals

Instruction Decode and Control PLA:

During normal operation, the INS807X repeatedly fetches and executes instructions stored in internal ROM/RAM or in external memory. Instruction access and execution are carried out under the control of an internal microprogram stored in a Programmable Logic Array (PLA). The INS807X instruction set may be modified by altering the contents of the PLA by a mask change. (PLA changes are possible on large volume orders only.) For further information on PLA modification call National Semiconductor's Microprocessor Division.

Bus Access Control

Before the INS807X can initiate data transfers with external peripherals or external memory, it must have access to the external address and data buses. Three of the INS807X I/O pins are associated with bus access control: NBREQ, NENIN and NENOUT. For simple stand-alone applications, the NENOUT signal can be ignored; the NENIN pin should be grounded or connected to NBREQ. The NBREQ pin must be connected to V_{CC} through an external pullup resistor.

Extended Bus Access Timing Considerations

There are four possible permutations of data and instruction accesses:

a. Instruction and data both internal
b. Instruction and data both external
c. Instruction internal, data external
d. Instruction external, data internal

For an external instruction, a number of microcycles equal to the number of bytes fetched by the instruction must be added to the total internal execution time.

For external data, a number of microcycles equal to the number of data bytes fetched must be added to the total instruction execution time.

The appropriate combination should be selected and the correct number of microcycles added to the execution time.

Exceptions to this general rule are:

a. Immediate data - treated as part of the instruction.

b. Increment, Decrement and Load - one data byte is accessed twice therefore, two microcycles must be added.

c. One byte call - three separate areas of memory are addressed.

   1) Single byte call
   1) Address
   3) Stack

Therefore, each area must be defined as internal or external.

NOTE: *Stack accesses must also be factored, as they are normally treated as data.*

External READ/WRITE Cycle Extension

As shown in *Figure 3*, the NHOLD signal may be set low following the leading edge of the READ strobe (NRDS) or the WRITE strobe (NWDS). Setting NHOLD low causes the INS807X to extend an external I/O (READ or WRITE) cycle. When an external I/O cycle is extended, the INS807X address, data and strobe lines are held active until the NHOLD signal is returned high. There is no restriction for the maximum duration of NHOLD in the low state. Thus, NHOLD can be utilized in a variety of ways, ranging from accommodation of slow memories/peripherals to single READ/WRITE cycle execution for software debug purposes. Timing parameters for the READ/WRITE cycle extension are listed in the AC Electrical Characteristics.

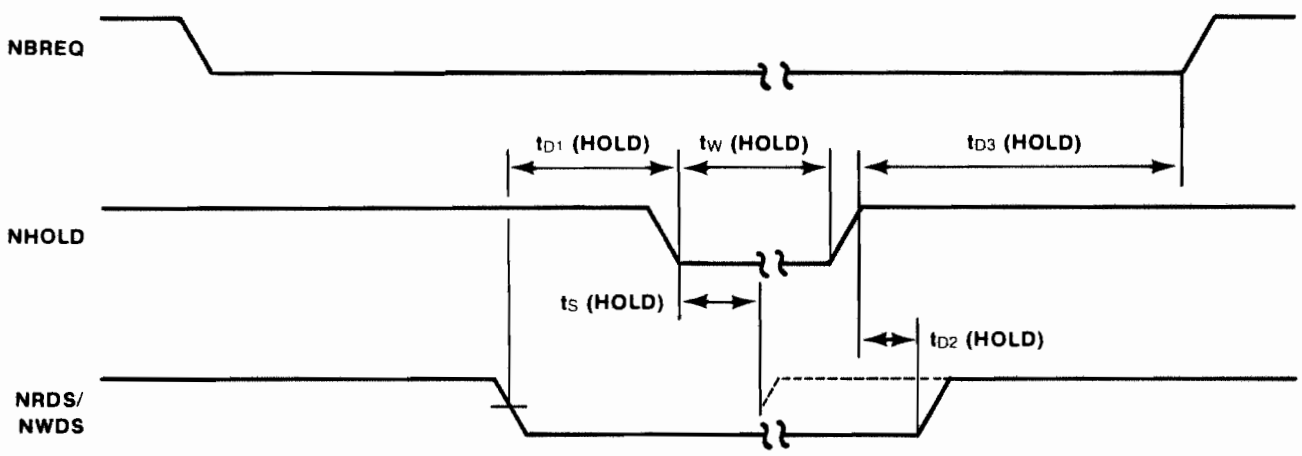


FIGURE 3. Extended Input/Output Timing Using NHOLD

Interrupt Servicing

When interrupts are enabled (IE = 1 in the Status register), pins SA/INTA and SB/INTB act as vectored interrupts. Both inputs are edge triggered and are activated by the high to low logic transition. (Edge triggered interrupts offer the advantage of not requiring an interrupt acknowledge signal to remove the interrupt service request, as required by level triggered interrupts.) Interrupt service requests are latched when they occur and are recognized prior to each instruction fetch. When an interrupt is recognized, the contents of the program counter are pushed onto the stack, saving the return address. The interrupt enable bit (IE) in the STATUS register is automatically reset, disabling further interrupts from being recognized, but interrupts occurring during this time will still be latched. The processor then fetches the next instruction from a predetermined memory location. The location for interrupt A is 0004_{16} and location for interrupt B is 0007_{16}. (Refer to *Figure 4.*) The instructions stored in these vector locations are normally BRANCH or JUMP instructions, causing the processor to begin execution of the appropriate service subroutine. The interrupt service subroutines may be of arbitrary length and may be located anywhere in memory.

Since the Interrupt Enable bit (IE) is automatically cleared at the start of interrupt service, Interrupt A cannot normally interrupt itself or Interrupt B, and vice versa, unless interrupts are re-enabled in the service subroutine. Since interrupt requests are latched, an interrupt request will never be lost even though it occurs during servicing of the other interrupt.

Interrupt service subroutines should always be concluded with an interrupt enable instruction (OR S = 01_{16}) immediately followed by a return instruction (RET). This sequence causes the processor to resume main program execution at the point where it was interrupted.

Interrupt Timing and Interrupt Priority

Interrupts are latched when they occur and are sampled prior to each instruction fetch. Thus, when an interrupt occurs during execution of an instruction, the instruction in process must be completed before the interrupt will be recognized. This results in an Interrupt Recognize Latency Time (IRLT), which is a function of the time at which the asynchronous interrupt occurs. The minimum value for the IRLT is 9 microcycles, and the maximum value is equal to the duration of the longest program instruction (in microcycles) plus 9 microcycles.

Interrupt A Always has Priority Over Interrupt B

If both interrupts occur during the IRLT, Interrupt A will always be recognized first, even though Interrupt B may have occured slightly earlier. In order for Interrupt B to be recognized before Interrupt A, Interrupt B must occur prior to Interrupt A by an amount greater than the maximum value of the IRLT.

If several interrupt requests occur on the same pin during the IRLT, they will be treated as a single interrupt request. If two interrupt requests occur on the same pin spaced apart by a time duration greater than the IRLT, they will be treated as two separate requests. For this case, if the second interrupt occurs during interrupt service, service will not be interrupted unless interrupts have been re-enabled by the interrupt service routine.

The Interrupt Service Latency Time (ISLT) is the time duration measured from the active edge of the interrupt request to the start of execution of the first instruction in the interrupt service routine. When interrupts are enabled, the ISLT is equal to the IRLT plus 8 microcycles or 5 microcycles, depending upon whether a JUMP or BRANCH instruction is stored at the vector memory locations 0004_{16} and 0007_{16}

MEMORY LOCATION	MEMORY CONTENTS	
0		
1	JMP	RESTART
2	XX	
3	XX	
4	JMP	INTERRUPT A
5	XX	
6	XX	
7	JMP	INTERRUPT B
8	XX	
9	XX	
A		

FIGURE 4. Interrupt Transfer Locations

INS807X Architecture and Internal Registers

As shown in the Block Diagram, INS807X contains the equivalent of 18, 8-bit registers; their functions are described below:

Program Counter: PC The program counter is a 16-bit register that contains the address of the instruction to be executed. The program counter is incremented before each instruction fetch. The program counter contents may also be replaced by the contents of a specified memory location if the last instruction was a branch, jump, or jump to subroutine instruction. This causes execution to continue from the program specified location rather than from the next sequential location. The PC may also be exchanged with the contents of the Extended Accumulator (EA).

Pointer Registers: (SP, P2, P3) The three pointers, SP, P2, and P3 are 16-bit registers that may be modified under program control. A pointer which has been loaded with a given address can reference up to 256 memory or I/O bytes without being modified. This is accomplished using pointer relative addressing.

The stack pointer, SP, may be used as a general purpose pointer and/or to point to the top of a RAM-resident stack that is implicitly referenced by the CALL, JSR, RET, PUSH, POP and PLI instructions. Under program control, this stack may be located anywhere in memory and may be of arbitrary length.

In addition to the stack referenced by the stack pointer (SP), pointers P2 and P3 may also be used to reference other stacks. This is accomplished using auto-indexed addressing.

Accumulator: AC The 8-bit Accumulator Register (AC) is the primary working register of the INS807X and contains the results of all single byte (8-bit) arithmetic and logic operations. Its contents may also be shifted, rotated, exchanged with the Extension Register (E) and moved on and off the stack using the PUSH and POP instructions.

Extension Register: E The 8-bit Extension Register (E) is used for temporary storage, to perform arithmetic and logic operations in conjunction with the Accumulator (AC) and combined with the accumulator to form the Extended Accumulator (EA). (For 16-bit operations.)

Double Precision Accumulator Register: EA The 16-bit Double Precision Accumulator Register (EA) is the primary double byte working register of the INS807X. It is formed by the concatenation of the 8-bit Accumulator Register (AC) and the 8-bit Extension Register (E). All double byte words are transferred between (EA) and memory in two single byte transfers.

Temporary Register: T The 16-bit Temporary Register (T) is used for temporary storage of double byte words. It is also used to hold the 16-bit multiplier and the 16-bit divisor in multiplication/division. The (T) register may be loaded directly from memory and its contents may also be transferred to/from the (EA) register.

Status Register: S The 8-bit Status Register (S) is used to store the following status bits: Carry/Link (CY/L), 2's Complement Overflow (OV), Sense B (SB), Sense A (SA), flags (F1, F2, F3) and the Interrupt Enable bit (IE).

The bit positions of the status register are shown below:

Bit positions

7	6	5	4	3	2	1	0	
CY/L	0V	SB	SA	F3	F2	F1	IE	Flags

(CY/L) goes high to indicate a carry output following execution of an addition/subtraction instruction.

Status register bits 1, 2 and 3 are buffered and connected to the flag output pins F1, F2, and F3. These pins are particularly useful for direct I/O control and for execution of serial I/O operations in conjunction with the SA and SB sense inputs.

SA and SB are Schmitt triggered buffered inputs which may be tested under software control. They form bits 4 and 5 of the Status Register and are read-only bits. All other Status Register bits may be both written to or read from using the AND S and the OR S instructions, respectively.

The Interrupt Enable bit (IE), in the Status Register may be set/reset under program control by the OR S and AND S instructions. When set high, IE enables interrupts A and B. When IE is low, both interrupts are disabled. Following a hardware reset via the NRST pin, IE is automatically set low, disabling both interrupts.

NOTE: *All Status Register bits except for SA and SB are latched. When set to a specified state under program control, they will remain in that state until again modified under program control.*

Instruction Register: This instruction register is not accessible to the programmer. During the fetch cycle of each instruction, this register is loaded with the 8-bit instruction op code retrieved from memory.

Auxiliary Register and Data I/O Register: These 8-bit registers are not accessible to the programmer. They are used for temporary storage of input/output bytes transmitted over the 8-bit bidirectional data bus.

Address Register: The 16-bit address register is not accessible to the programmer. It is used for temporary storage of the 16-bit address transmitted during a memory READ/WRITE cycle.

Arithmetic Logic Unit (ALU): The Boolean operations provided by the ALU include AND, OR, and XOR. The ALU also performs increment, decrement, add, subtract, multiply, divide, shift and rotate operations.

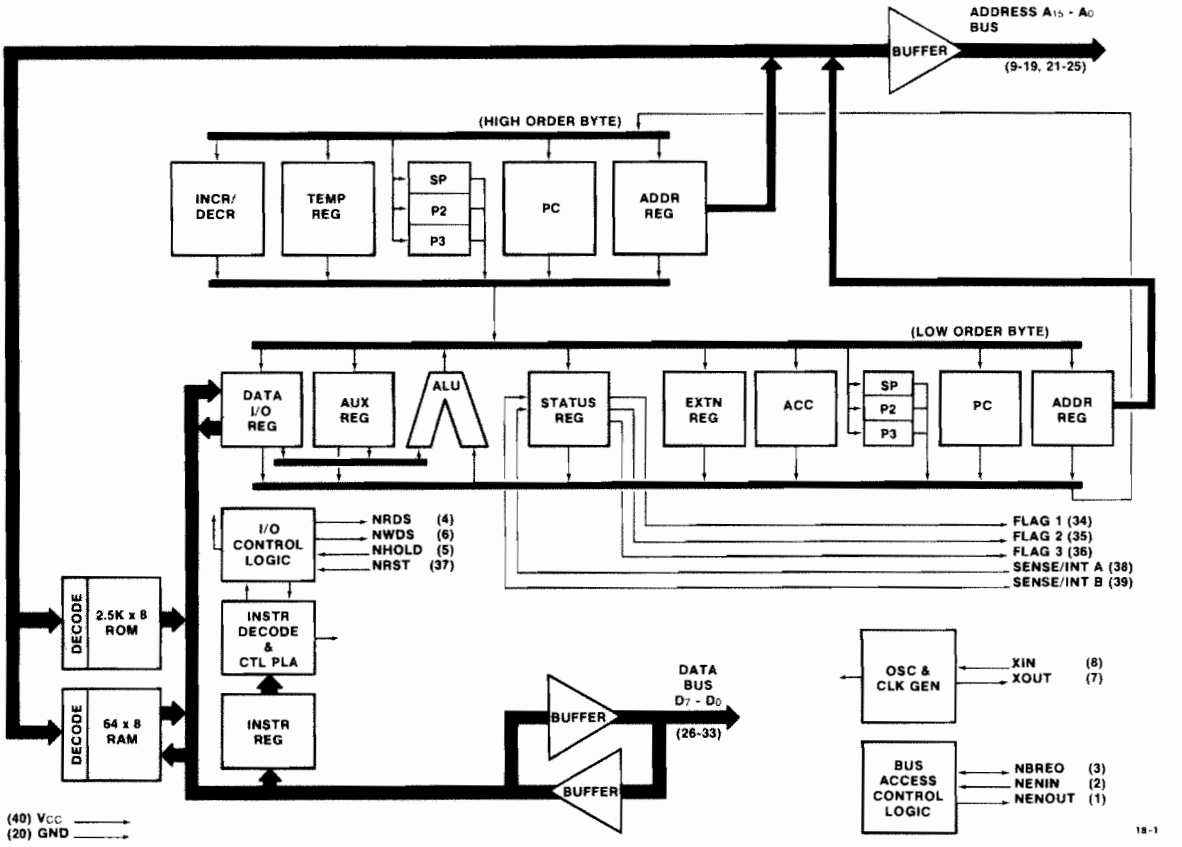


FIGURE 5. INS8070-Series Block Diagram

INSTRUCTION SET

The INS870X instruction set consists of 192 instructions which can be grouped into nine general catagories;

1) Register Load/Store Instructions.
2) Register to Register Exchange Instructions.
3) Arithmetic Operations: Add, Subtract, Multiply and Divide.
4) Boolean Operations: AND, OR, Exclusive OR.
5) Shift and Rotate Instructions.
6) Stack PUSH/POP Instructions.
7) Program Counter Modification Instructions.
8) Text Processing Instructions.
9) Miscellaneous Instructions.

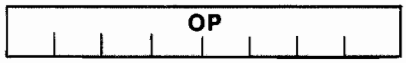
A listing of the complete instruction set is provided in *Appendix A* of the User's Manual. In the appendix, the symbol OP refers to the hexadecimal OP code for each instruction. The symbol BY refers to the number of memory bytes required for each instruction. The symbol RW refers to the number of READ plus WRITE operations required to execute each instruction. Note that OP, BY and RW depend upon the addressing mode specified.

The symbol MC refers to the number of microcycles required to execute each instruction. (One microcycle equals 1 μsec when the processor is operating at an oscillator frequency of 4 MHz.) The numbers appearing under the symbol MC apply to the case where the op code and all operands are fetched/stored from internal ROM or RAM. When external memory is accessed additional microcycles must be added, because external memory I/O requires an additional microcycle per READ or WRITE operation. In addition to data READ and WRITE, external READ operations to fetch the op code and displacement (if required) must be taken into consideration.

INSTRUCTION FORMAT

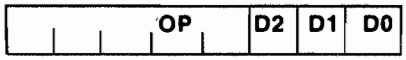
All INS807X instructions fall into three catagories: single, double, and triple byte. *Figure 6* indicates the basic format for all three instruction catagories.

One Byte Instructions



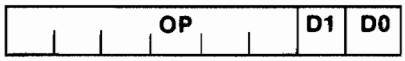
OP completely specifies the operation to be performed. No further memory references are required.

Two Byte Instructions

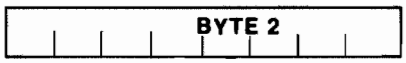


OP specifies the operation to be performed. The addressing mode is determined by bits D0, D1 and D2. For all addressing modes except immediate BYTE 2 represents a displacement value between $+127_{10}$ and -128_{10} that is used to calculate the effective memory address. When the immediate mode is specified BYTE 2 contains data.

Three Byte Instructions



If OP specifies an operation in which D0 and D1 specify a pointer (PC, SP, or P3), bytes 2 and 3 represent a 16-bit address.



If OP specifies an operation that does not involve a pointer, bytes 2 and 3 represent 16 bits of immediate data. Byte 2 is the least significant byte, and Byte 3 is the most significant byte.

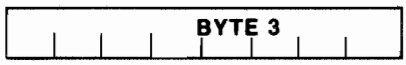


FIGURE 6. Instruction BYTE Formats

SPECIALIZED INSTRUCTIONS

NOTE: *The number of microcycles (MC) and read/write (RW) operations for the SSM and BND instructions are not listed in the instruction set (refer to Appendix A) because the values of these numbers are data dependent.*

Search and Skip if Character Matched (SSM) Instruction

The SSM instruction searches memory for a character that matches the one present in the accumulator, using indexed addressing. The address where the search begins may be specified by pointers P2 or P3. The number of memory locations searched is 256. The search is conducted by using the specified pointer to indicate the address of the character to be tested. The pointer is automatically incremented by 1 after each test. If a match is found within the allowable 256 searches, the next two program bytes will be skipped. In this case, the instruction is concluded with the specified pointer pointing to an address one greater than that for which the match occurred. If a match is not found within the allowable 256 searches, the next instruction in sequence will be executed. In this case, the specified pointer will have been incremented by one less than the number of searches allowed (255).

Branch If Not Digit (BND) Instruction

The two byte BND instruction tests the ASCII character present in the accumulator to determine whether or not it represents the ASCII equivalent of a decimal digit (30_{16} -39_{16}). If it does not, a PC relative branch is executed using the second byte of the instruction as displacement. In this case, the contents of the accumulator are not altered. If the accumulator does contain the ASCII equivalent of a decimal digit, the accumulator contents are converted to the 8-bit binary equivalent of the decimal digit (00_{16} - 09_{16}).

One Byte Call Instructions

The CALL and JSR instructions are similar in that both allow the user to invoke a subroutine. A major difference, however, is that the CALL instruction occupies only one byte, whereas the JSR instruction occupies three bytes. The CALL instruction utilizes indirect addressing and allows the user to invoke up to 16 subroutines. Each of the 16 subroutines may be of arbitrary length and located anywhere in memory. The two byte pointer address for each subroutine is stored in two consecutive memory locations, starting at 32_{10} (20_{16}). The values of the subroutine starting addresses stored in these indirect locations should be one less than the memory location where the subroutine actually begins. Subroutines invoked via a one byte CALL are exited in the normal manner using a RET instruction.

Multiply Instruction

The INS807X multiplication instruction (MULT) multiplies a signed 16-bit multiplicand by a signed 16-bit multiplier, generating a signed 32-bit product. The signed 16-bit multiplicand may be positive or negative and is stored in the double byte accumulator EA. (EA = the concatenation of register E and A.) The signed 16-bit multiplier must be positive and is stored in the 16-bit register T. The signed 32-bit product is stored in the EA and T registers, replacing the original multiplier and multiplicand. The most significant 16 bits of the product are stored in the EA register, and the least significant 16 bits of the product are stored in the T register. In shorthand notation, the multiplication instruction may be written as follows:

$$(EA) \leftarrow (EA) * (T)_{31:16}$$

$$(T) \leftarrow (EA) * (T)_{15:0}$$

NOTE: *The subscripts represent bit positions of the product.*

The multiplication instruction is executed in only 37 micro-seconds at an oscillator frequency of 4 MHz.

Divide Instruction

The INS807X divide instruction divides a 16-bit dividend by a 16-bit divisor, generating a 16-bit quotient. Since the divide instruction is an integer divide, the magnitude of the dividend must be greater than the magnitude of the divisor, which must be non-zero. The 16-bit dividend can be an unsigned number in the range 0_{10} - $65{,}535_{10}$ or a signed positive number in range 0_{10} - $32{,}767_{10}$. The divisor can be an unsigned number in the range 0_{10} - $32{,}767_{10}$ or a signed, positive number in the range 0_{10} - $32{,}767_{10}$. Thus, the divide instruction will divide a signed dividend by a signed divisor, when both quantities are positive. It will also divide an unsigned dividend in the range 0_{10} - $65{,}535_{10}$ by an unsigned divisor in the range 0_{10} -$32{,}767_{10}$.

Before division can be performed, a 16-bit dividend must be stored in the EA register and a 16-bit divisor must be stored in the T register. After division has been executed, a 16-bit quotient is stored in the EA register, replacing the original dividend; the divisor which has been stored in the T register is lost.

The division instruction is executed in only 42 microseconds at an oscillator frequency of 4 MHz.

Memory Allocation

Figure 7 shows INS807X memory allocation, (decimal and hexadecimal address) for the National Semiconductor 70-Series family.

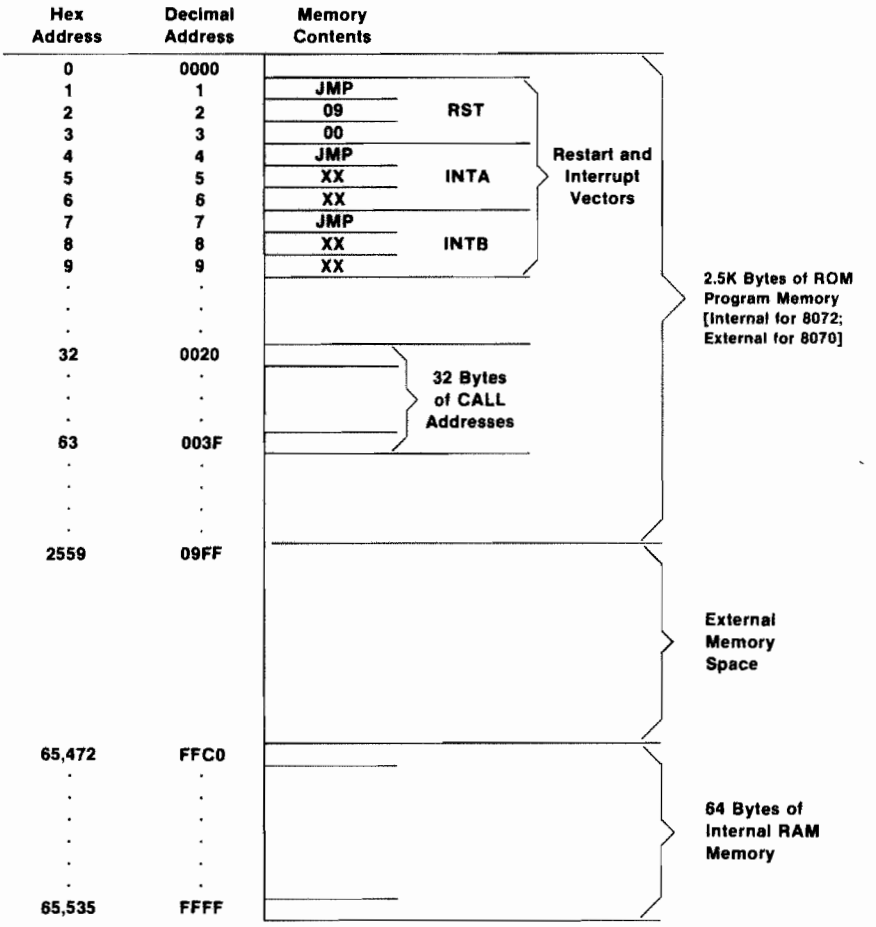


FIGURE 7. INS807X Memory Allocation

INITIALIZATION

The recommended power-up initialization sequence for the INS807X is as follows:

1. Apply power (GND and V_{CC}) and set NRST low. When NRST goes low the following occurs:

   a) Any in-process operations are automatically aborted.

   b) The READ/WRITE (NRDS/NWDS) strobes, address bus and data bus are set to the TRI-STATE (high Z) condition.

   c) The Program Counter (PC), Stack Pointer (SP), and Status Register (S) are automatically cleared.

2. Ample time should be allowed for the oscillator and internal clocks to stabilize. The minimum time that NRST must remain low is 4 T_C, where T_C is the period of one microcycle. (NOTE: *For a crystal controlled oscillator, 250 mS are typically required for crystal warmup and stabilization.*)

3. To commence program execution, NRST must be set high. The first instruction will be fetched from location 0001_{16}. This will occur within 13 T_C after NRST has gone high.

Normal program execution will continue as long as NRST remains high. In order to allow this signal to have slow rise/fall times without any adverse affects, NRST is buffered by a T^2L compatible Schmitt trigger. This allows the inexpensive RC power-on reset circuit of *figure 8* to be used to initialize the INS807X.

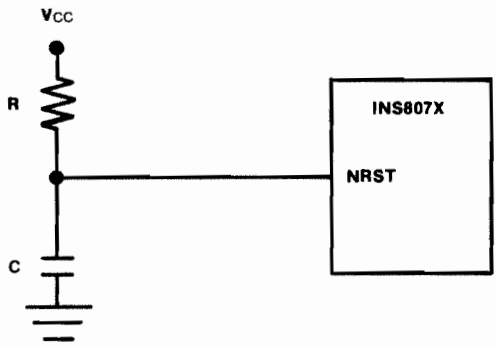


FIGURE 8. Reset Implementation 18-6

Interfacing With External Peripherals

Data are transferred on the INS807X external data bus during execution of external memory reference instructions. This enables the INS807X to use its entire memory reference class of instructions, including arithmetic and logic, to implement input/output (I/O) to external peripherals. Since all memory reference instructions are available for peripheral I/O, efficient service routines can be written that execute faster and require fewer memory bytes.

The INS807X can implement serial I/O efficiently via its two sense inputs (SA/INTA, SB/INTB) and three flag outputs (F1, F2, F3). Using these, two independent serial I/O channels can be implemented under software control using one sense input and one flag output per channel. Devices which can use this feature include teletypewriters, terminals, and modems.

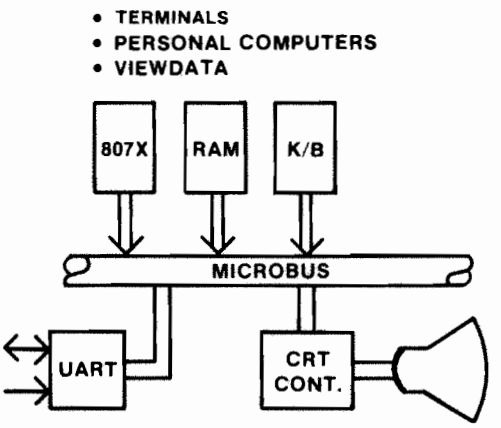
APPLICATIONS

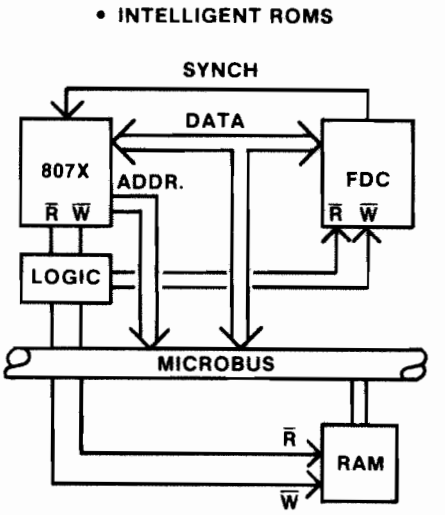
The 70-Series family of microprocessors is especially useful for the following applications.

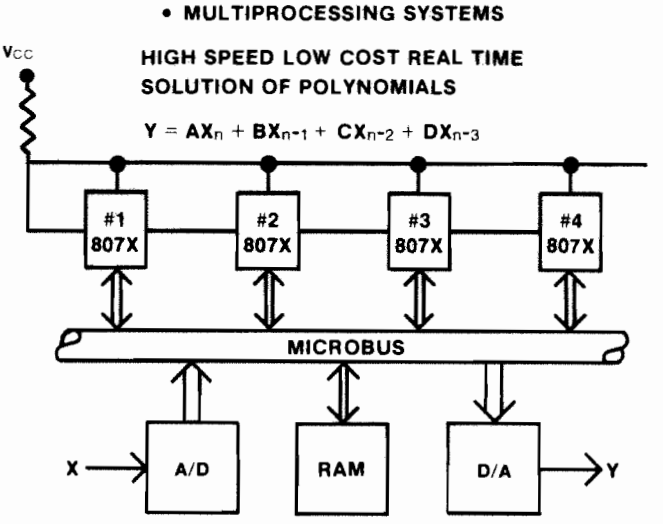
Terminals, personal computers and Viewdata sets. (These three applications contain identical requirements for data display via a CRT.) System cost is drastically reduced by simplified buffering and synchronization requirements between the display controller, system RAM and CPU. Typically a dumb terminal may be implemented with less than 18 IC's.

Intelligent ROMs - The INS807X system shown fulfills two functions: (1) generate addresses for the floppy disk controller (2) control floppy disk controller. This provides a system that does not require a DMA controller or a processor-specific firmware package for the file processing. This system therefore releases more memory for user programs and allows increased flexibility.

Multiprocessing Systems - Through the use of the on-board executable RAM, and by taking advantage of the high-speed 16-bit arithmetic operations, real-time solutions of polynomials become extremely simple and economical.







18-12

INPUT MEDIUM:

EPROM

PAPER TAPE

EPROM:

2716 Total No. of EPROMS

2708

IMPORTANT - EPROM LABELLING

Only one customer program may be included in a single order. The following method must be used to identify the EPROMs comprising a program.

a) The EPROMs used for storing a custom program are designated as shown:

2716:	BLOCK A	0-2047
	BLOCK B	2048 - 2559
2708:	BLOCK A	0 - 1023
	BLOCK B	1024 - 2047
	BLOCK C	2048 - 2559

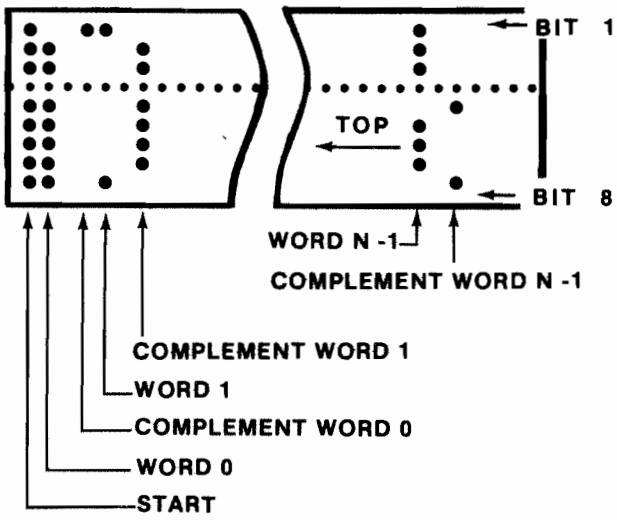
b) All EPROMs must be labelled (stickers, paint, etc) with this block designation plus a customer identification number. For example a 2.5K custom program contained in two 2716's would be labelled:

Customer I.D.	Customer I.D.
A	B
0-2047	2048-2559

Paper Tape

Tapes may only be submitted in binary complement format. The following information should be written on the paper tape.

Company Name
Company I.D. No.
NSC Part No.
A Punch = ("1" or "0")
This is logic (POS or NEG)



NOTE 1: Tape must be blank except for the data words.
NOTE 2: Tape must start with a rubout character.
NOTE 3: Data is comprised of two words, the first being the actual Data and the second being the complement of the data.

Binary Complement Format

Verification

The customer can use software (the listing) or hardware (EPROMs) to verify the program. He will receive a verification listing and a set of EPROMs programmed and tested with tapes generated by the NSC Mask Programming System (MPS). He will be asked for a GO/NO GO response within a week after receipt of the listing and EPROMs.

Verification Listing

The verification listing has five sections:

1. A cover sheet with provision for "STOP, DO NOT PROCEED" or "VERIFICATION CERTIFIED" signatures.
2. A description of the logic designations and assumptions used to process the data.
3. A listing of the data submitted by the customer.
4. An error summary.
5. A definition of the standard logic definitions for the ROM and the reduced form of the data. This list shows the output word corresponding to each address coded three ways - binary, octal, and hexadecimal.

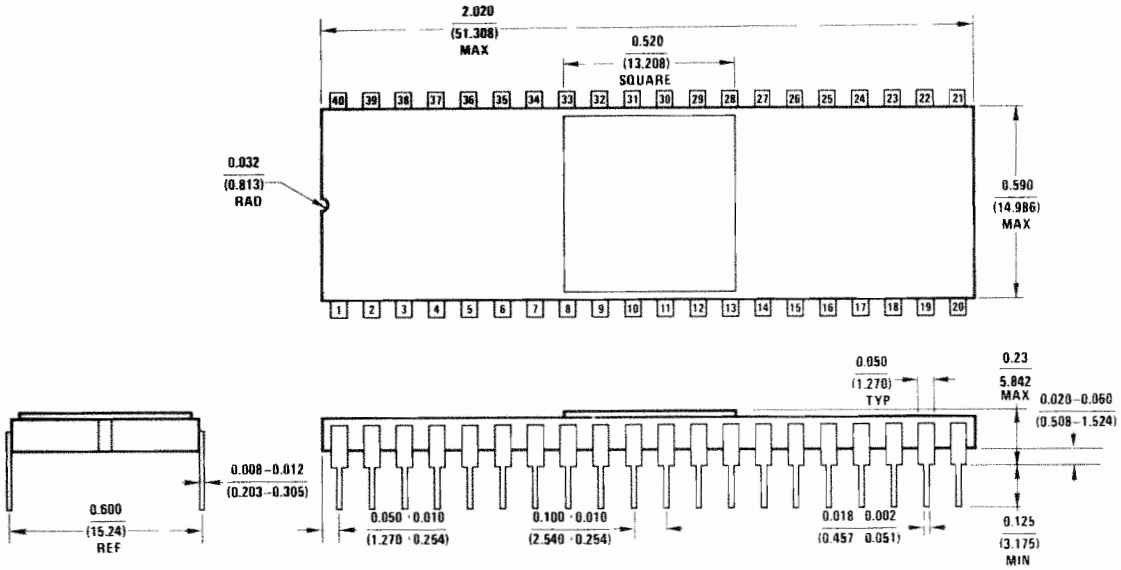
Ordering Information for Custom Programmed Parts

The following information must be submitted with each customer microcomputer program. An order will not be processed unless it is accompanied by this information. This form acts as a Traveler from Customer through Customer Service to ROM programming. Please retain a copy of this form to compare against the verification listing. The form will be sent back to the customer by Customer Service.

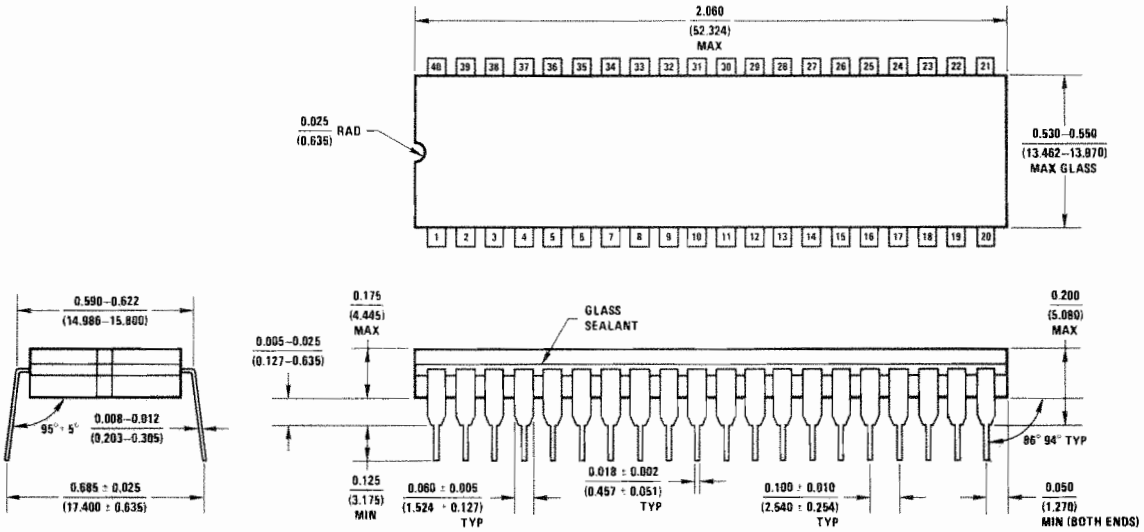
			National Microcomputer Part Number
			ROM Letter Code (National Use Only)
Name			Date
Address			Customer Print or I.D. No.
City	State	Zip	Purchase Order No.
Telephone ( )			Name of person National can contact (Print)
Authorized Signature			Date

Physical Dimensions

inches (millimeters)



Ceramic Dual-In-Line Package (D)
Order Number INS8070D



Plastic Dual-In-Line Package (J)
Order Number INS8070 or INS8072XXX/J

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara California 95051
Tel (408) 737-5000
TWX (910) 339-9240

National Semiconductor GmbH
Eisenheimerstrasse 61/II
8000 München 21
West Germany
Tel (089) 576091
Telex 05-22772

NS International Inc., Japan
Miyake Building
1-9 Yotsuya Shinjuku-ku 160
Tokyo Japan
Tel (03) 355-3711
TWX 232-2015 NSCJ J

National Semiconductor (Hong Kong) Ltd
8th Floor
Cheung Kong Electronic Bldg
4 Hing Yip Street
Kwun Tong
Kowloon Hong Kong
Tel 3-899235
Telex 73866 NSEHK HX
Cable NATSEMI

NS Electronics Do Brasil
Avda Brigadeiro Faria Lima 844
11 Andar Conjunto 1104
Jardim Paulistano
Sao Paulo Brasil
Telex
1121008 CABINE SAO PAULO

NS Electronics Pty. Ltd.
Cnr Stud Rd & Mtn Highway
Bayswater Victoria 3153
Australia
Tel 03-729-6333
Telex 32096

National does not assume any responsibility for use of any circuitry described no circuit patent licenses are implied and National reserves the right at any time without notice to change said circuitry

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
Tel: (408) 737-5000
TWX: (910) 339-9240

National Semiconductor
District Sales Office
345 Wilson Avenue, Suite 404
Downsview, Ontario M3H 5W1
Canada
Tel: (416) 625-7260

Mexicana de Electronica
Industrial S. A.
Tiacoquemecatl No. 139-401
Esquina Adolfo Prieto
Mexico 12, D.F.
Tel: 575-78-68, 575-79-24

NS Electronics Do Brasil
Avada Brigadeiro Faria Lima 844
11 Andar Conjunto 1104
Jardim Paulistano
Sao Paulo, Brasil
Telex: 1121008 CABINE SAO PAULO

National Semiconductor GmbH
Elsenheimerstrasse 61/2
8000 Munchen 21
West Germany
Tel (089) 576091
Telex: 05-22772

National Semiconductor (UK) Ltd.
301 Harpur Centre
Horne Lane
Bedford MK40 1TR
United Kingdom
Tel: 0234-47147
Telex: 826 209

National Semiconductor
789 Ave Houba de Strooper
1020 Bruxelles
Belgium
Tel: (02) 4783400
Telex: 61007

National Semiconductor Ltd.
Vodroffsvej 44
1900 Copenhagen V.
Denmark
Tel: (01) 356533
Telex: 15179

National Semiconductor France
Expansion 10000
28, rue de la Redoute
92-260 Fontenay aux Roses
France
Tel. 660 81 40
Telex: NSF250956

National Semiconductor SRL
Via Alberto Mario 26
20149 Milano
Italy
Tel (02) 469 28 64/469 24 31
Telex: 332835

National Semiconductor Sweden
Algrytevagen 23
S-127 32 Skarholmen
Sweden
Tel: (8) 970835
Telex. 10731

NS International Inc., Japan
Miyake Building
1-9 Yotsuya, Shinjuku-Ku 160
Tokyo, Japan
Tel (04) 355-5711
TWX: 232-2015 JSCJ-J

National Semiconductor (Hong Kong) Ltd.
8th Floor
Cheung Kong Electronic Bldg.
4 Hing Yip Street
Kwun Tong
Kowloon, Hong Kong
Tel: 3-899235
Telex. 73866 NSEHK HX
Cable: NATSEMI

NS Electronics Pty. Ltd.
Cnr Stud Rd. & Mtn Highway
Bayswater, Victoria 5153
Australia
Tel: 03-729-6333
Telex: 32096

National Semiconductor (Pty.) Ltd.
No 1100 Lower Delta Road
Singapore 3
Tel: 2700011
Telex: NAT SEMI RS 21402

National Semiconductor (Taiwan) Ltd.
3rd Floor, Apollo Bldg.
No. 218-7 Chung Shiao E. Road
Sec. 4, Taipei, Taiwan
P O Box 68-332 Taipei
Tel 7310393-4 7310465-6
Telex: 22837 NSTW
Cable: NSTW Taipei

National Semiconductor (Hong Kong) Ltd.
Korea Liaison Office
Rm. 607, 615 Seoul Bldg.
6, 2 Ka Huehyun-Dong Bldg.
C. P O. Box 7941
Jung Ku, Seoul 100
Korea
Tel: 28-4746

©1979 National Semiconductor Corp